

SCIENTIFIC XPL REFERENCE MANUAL
for
New England Digital Corporation's
ABLE SERIES COMPUTERS

June 1987

NEW ENGLAND DIGITAL CORPORATION
49 North Main Street
P.O. Box 546
White River Junction, Vermont 05001

Copyright 1987 - New England Digital Corporation
White River Junction, Vermont

SCIENTIFIC XPL REFERENCE MANUAL

Introduction to Scientific XPL	7
Section I - Constants and Variables	13
Data Types	13
Constants	14
Variable Identifiers	16
Variable Declarations	17
Literals	18
Section II - Terminal Input and Output	19
PRINT, SEND Statements	19
CHARACTER, OCTAL, STRING Formats	20
INPUT Statement	21
LINPUT Statement	22
Section III - Operators and Expressions	25
Arithmetic Operators	25
Relational Operators	29
Unsigned Relational Operators	30
Logical Operators	31
Bit Operators	32
Assignment Statement	35
Numeric Conversions	35
Precedence and Order of Evaluation	36
Section IV - Flow of Control	37
Compound Statements	37
IF Statement	38
DO WHILE Loops	39
Iterative DO Loops	40
DO CASE Statement	42
Statement Labels	43
GOTO Statement	44
Section V - Arrays and Pointers	45
Arrays	45
Character Strings	47
BYTE, PBYTE Routines	48
DATA Declaration	49
ADDR Function	50
CORE Array	50
LOCATION Function	51

Section VI - Procedures	53
Program Structure	53
Procedures	55
Passing by Value	58
Passing by Reference	59
Functions	62
Block Structure and Scope	64
Variable Storage Classes	69
Recursive Procedures	70
Swapping Procedures	71
Forward Reference Calls	72
Section VII - Modules and Libraries	73
INSERT Statement	75
ENTER Statement	76
Modules	78
Public Variables and Procedures	80
External Variables and Procedures	81
Libraries	82
An Example Module	86
Section VIII - Hardware Manipulation	89
READ Function	89
WRITE Statement	90
Assembly Language Programming	91
Section IX - Interrupt and Exception Processing	93
WHEN Statement	93
ENABLE, DISABLE Statements	93
Interrupt Processing	94
Exception Processing	97
INVOKE Statement	97

REFERENCE APPENDICES

Appendix A - Built-In Functions	101
Storage Device I/O	101
XPL Built-In Functions	104
Appendix B - Compilation Control	135
Compile-time Options	135
CONFIGURATION Statement	136
RAM, PDL Statements	137
EOF Symbol	137
Appendix C - Summary of XPL	139
Appendix D - Compiler Error Messages	147
Appendix E - Internal Representations	159
Appendix F - Memory Layout	163
Appendix G - D4567 Operation	171
Index	173

This manual describes the Scientific XPL language for version 6.00 and beyond (June 1987).

The material in this manual is for informational purposes and is subject to change without notice.

New England Digital Corporation assumes no responsibility for any errors which may appear in this manual.

Introduction to Scientific XPL

New England Digital's Scientific XPL is a high-level structured programming language that is used to write real-time programs for ABLE computers. Scientific XPL is an efficient and easy to use language that has been specifically designed for high speed scientific and industrial applications. Special features are included in Scientific XPL that allow data collection, bit manipulation, and other real-time functions to be easily performed. These features work together to provide an effective software tool that can be used to quickly and reliably solve challenging software problems.

Scientific XPL is one of a breed of modern programming languages that has become very popular with a wide range of computer users. A minimal subset of PL/I, Scientific XPL is a block-structured language that supports many variable types and includes powerful control structures such as IF statements and DO WHILE loops. These features of Scientific XPL make it vastly superior to older languages such as FORTRAN, COBOL, and BASIC.

Modern concepts of structured language programming are combined in Scientific XPL with the high speed and efficiency produced by a three pass optimizing compiler. This provides the programmer with the programming ease of a good high level language, yet also provides the ability to process data with an efficiency that on other systems is only available with assembly language programs. Scientific XPL has been used successfully in many applications where execution speed, program development time, program maintainability, and software reliability are important.

The efficiency of Scientific XPL derives from the three pass architecture that is used in the Scientific XPL Compiler. The three pass approach has many advantages when compared with the interpretive or single pass approach used on other systems. Since all of the symbol table processing is performed during the first pass, a great deal of memory is made available during the second pass for a powerful optimization routine. This routine creates a tree structure for each arithmetic expression which is then carefully inspected for common subexpressions, use of the multiply-divide unit, and so forth. Many Scientific XPL statements are compiled into a single directly executable machine instruction that takes on the order of one microsecond to perform.

Basic Features of Scientific XPL

A Scientific XPL program is a list of program statements. Most statements direct the computer to calculate an arithmetic expression, test for certain logical conditions, or perform input and output. Other statements allow the programmer to control the allocation of storage, create simple textual substitutions (literals), and define procedures. Scientific XPL is a block-structured language; procedures can contain further declarations which control storage allocation and define other procedures.

The procedure definition facility of Scientific XPL allows modular programming, so that a program can be divided into sections (e.g, numeric input, conversion from binary to decimal forms, or printing output messages). Each of these sections can be written as a procedure. Such procedures are conceptually simple, easy to formulate and debug, and easily incorporated into a large program. They can form a basis for a procedure library, if a family of similar programs is being developed.

Scientific XPL operates with two basic data types: fixed point and floating point numeric values. Fixed point numbers are stored inside the computer in a single 16-bit word of memory and are normally treated as signed integers in the range of -32,768 to +32,767 (narrower ranges apply in certain situations such as when performing arithmetic comparisons). A floating point variable is stored in two locations of memory and is used to represent a number in the range of (plus and minus) 5.50 E-20 to 9.00 E18. Two other data types are also available that are derived from the fixed point type: boolean and pointer. One-dimensional arrays (vectors) can be declared to let one variable name represent many data elements.

Executable statements specify the computational processes that are to take place. To achieve this, arithmetic, logical, and comparative (relational) operators are defined for variables and constants of all data types. These operators and operands are combined to form expressions which resemble those of elementary algebra. For example, the expression

$$x*(y - 3)/r$$

represents this calculation: the value of X multiplied by the quantity Y - 3, divided by the value of R.

Expressions are included in many Scientific XPL statements. A simple statement is the assignment statement, which computes a result and stores it in a memory location defined by a variable name. The assignment statement:

$$q = x*(y - 3)/r$$

first causes the evaluation of the expression to the right of the equals sign, as described above. Then the result of this computation is stored in the memory location associated with the variable name Q.

Other statements perform conditional testing and branching (IF), loop control (DO, DO WHILE), procedure invocation with parameter passing (CALL), interrupt processing (WHEN), and terminal input and output (INPUT, LINPUT, PRINT).

A particularly important feature of Scientific XPL is its real-time input/output capability. The READ function and WRITE statement transfer 16-bit quantities to and from interface modules connected to the system. In most cases, the data transfer occurs in a single machine instruction, requiring on the order of one microsecond to perform. The interface modules are each identified by a unique device address. The ABLE Series architecture allows for 256 devices.

A method of automatic text substitution (more specifically, a compile-time macro facility) is also provided in Scientific XPL. Symbolic names can be declared to be equivalent to an arbitrary sequence of characters. As each occurrence of the symbolic name is encountered by the compiler, the declared character sequence is substituted, so the compiler actually processes the substituted character string instead of the symbolic name. Such LITERAL declarations can be used to noticeably increase program readability and provide for self-documenting program statements.

A sample program that shows many features of Scientific XPL appears on the following pages. Notice that XPL statements end with a semicolon, and that spaces can be used throughout a program to improve readability. Comments begin with the character pair /* and end with */, as in the following example:

```
/* This is a comment, not a program statement */
```

```

1  /* Statistics Program
2
3  The following program will compute the average,
4  standard deviation and median of a list of
5  floating point numbers.
6
7  Vector data structure:
8  list(0) = n = number of elements in array
9  list(1) ... list(n) = vector values
10 */
11
12 dcl program_date data ('1 May 1987'); /* date of this program version */
13
14 dcl forever lit 'while (true)'; /* loop control */
15
16 dcl vector_length lit '200'; /* number of vector elements */
17
18 /* LEN returns the length of the vector, which is stored in the first
19 word of the vector (list (0)). */
20
21 len: proc (list) returns (fixed); /* compute length of vector */
22   dcl list floating array; /* array to process */
23
24   return int (list (0)); /* return length of vector */
25 end len;
26
27 /* AVERAGE adds all the vector elements and then finds the average
28 value of the vector. */
29
30 average: proc (list) returns (floating); /* compute average of array */
31   dcl list floating array; /* array to process */
32   dcl sum floating; /* local variables */
33   dcl i fixed;
34
35   sum = 0; /* initialize sum */
36
37   do i = 1 to len (list); /* do for all values in the vector */
38     sum = sum + list (i); /* sum values */
39   end;
40
41   return (sum/len (list)); /* return average to main program */
42 end average;
43
44 /* STANDARD_DEVIATION computes the standard deviation of the vector. */
45
46 standard_deviation: proc (list) returns (floating); /* computes the standard deviation of vector */
47   dcl list floating array; /* array to process */
48   dcl sum floating; /* sum of array elements */
49   dcl ave floating; /* average of array elements */
50   dcl i fixed;
51
52   if len (list) < 2 then do; /* error - print message and stop */
53     print 'Error in standard deviation - less than two vector elements';
54     call exit (0); /* exit the program */
55   end;
56
57   sum = 0; /* initialize sum */
58   ave = average (list); /* get the average */
59
60   do i = 1 to len (list); /* do for all vector values */
61     sum = sum + (list (i) - ave)*(list (i) - ave);
62   end;
63
64   return (sqr (sum/(len (list) - 1)));
65 end standard_deviation;
66

```

```

67  /* $page */
68
69  /* MEDIAN sorts the list, and returns the middle value. Not the
70     fastest algorithm but it works. This algorithm no faster than
71     O(n log n) but known algorithms can do it in O(n) time. */
72
73  median: proc (list) returns (floating); /* find median of vector */
74      dcl list floating array; /* array to process */
75
76      /* This SORT procedure uses a selection sort - replace it with
77         a faster algorithm later if needed. */
78
79      sort: proc (list); /* sort a vector */
80          dcl list floating array; /* array to process */
81          dcl smallest fixed; /* index of current smallest element */
82          dcl temp floating; /* temporary storage value */
83          dcl (i, j, n) fixed;
84
85          n = len (list); /* number of elements to sort */
86
87          do i = 1 to n - 1; /* do one pass from each array location */
88              smallest = i; /* assume first is smallest */
89
90              do j = i + 1 to n; /* check the rest of the elements */
91                  if list (j) < list (smallest) then smallest = j; /* if it's smaller, save it */
92              end;
93
94              /* place smallest in position I */
95              temp = list (i); list (i) = list (smallest); list (smallest) = temp;
96          end;
97      end sort;
98
99      call sort (list); /* sort the vector */
100     return (list ((len (list) + 1)/2)); /* return the middle element */
101
102 end median;
103
104 /* USER_INPUT gets all the vector values from the user. */
105
106 user_input: proc (list); /* get the vector values from the terminal */
107     dcl list floating array; /* array to process */
108     dcl (n, i) fixed;
109
110     print; print 'Enter the length of the data list';
111     input n; /* get the length */
112
113     if n > 0 then do; /* do only if there is something to input */
114         print 'Please enter data elements when prompted.';
115
116         do i = 1 to n; /* get all the values */
117             print 'Value';
118             input list (i); /* put the value in the array */
119         end;
120         list (0) = n; /* save length of vector in first slot */
121         print; /* print a blank line */
122     end;
123     else call exit (0); /* exit the program if no length specified */
124 end user_input;
125
126
127 /** MAIN PROGRAM **/
128
129 dcl vector (vector_length) floating array; /* vector to process */
130
131 print 'Standard Deviation Program - ', string (program_date);
132
133 do forever; /* infinite loop */
134     call user_input (vector); /* input vector values */
135     print 'Average = ', average (vector); /* print the average */
136     print 'Median = ', median (vector); /* print the median */
137     if len (vector) >= 2 /* must have at least two elements to do this */
138     then print 'Standard deviation = ', standard_deviation (vector);
139 end;

```


Section I - Constants and Variables

Numeric quantities in XPL can be either constants or variables. A constant always has the same value, while a variable's value can change during execution of the program. The expression:

$$x*(y - 3)/r$$

involves the constant 3 and the variables X, Y, and R. This section will describe the use of constants and variables in detail.

Data Types

Scientific XPL has four basic data types: FIXED, FLOATING, BOOLEAN, and POINTER. All of these types are stored in memory in a binary format. ALE computers are 16-bit computers, which means that each location of memory can store 16 bits (a bit is a binary digit) of information. Each fixed point, boolean, or pointer data element is in a 16-bit, single word location; each floating point data element is in a 32-bit, double word encoded form.

Fixed point numbers can be interpreted as signed integers in the range of -32,768 to +32,767 or as unsigned integers in the range of 0 to 65,535. Signed integers are used mostly in arithmetic expressions while unsigned integers are often used as array indexes. Both signed and unsigned integers are processed identically inside the computer; the difference lies in the user interpretation of the bit patterns stored in memory. Unsigned integers allow the processing of larger numbers in a single location of memory, but do not have the capability to represent negative quantities. Some of the arithmetic operators available in Scientific XPL such as multiplication, division, and the relational operators are designed to be used strictly with signed quantities. These limitations will be explained in detail later on.

Floating point numbers are signed reals in the range of plus and minus 5.5 E-20 to 9.0 E18.

Booleans are logic values restricted to TRUE and FALSE. This data type is most often used for flags. In Scientific XPL, all even numbers evaluate to false and all odd numbers evaluate to true. Presently, BOOLEAN is an alias for fixed, TRUE is the number one, and FALSE is the number zero.

Pointers are memory addresses and thus are unsigned integers in the range of 0 to 65,535. Pointers usually contain the address of another variable. They are presently defined to be fixed point numbers. The constant NULL is provided to distinguish between active and inactive pointers. The actual value for NULL is zero.

The internal representation of all data types is discussed in the appendix "Internal Representations".

Constants

A constant is a value known at compile-time which does not change during execution of the program. Both numeric constants and string constants are supported.

Fixed point constants can be expressed in decimal, octal, or hexadecimal form. Decimal values are specified by a sequence of up to five digits that optionally start with a minus sign (-) indicating a negative quantity:

```
25
-14
10456
```

Octal constants are specified by a sequence of up to six digits (zero through seven only) enclosed in quotation marks:

```
"002476"
"773"
"177777"
```

Hexadecimal constants are specified by an H followed by a sequence of up to four digits (or the letters A through F), all enclosed in quotation marks:

```
"H0001"
"Ha1b"
"HFFFF"
```

Floating point constants are specified by an optional minus sign followed by up to eight digits, optionally followed by a decimal point and up to eight more digits:

```
3.14159
-12.0
152.77124455
.0001
```

Although a decimal point is not required in a floating point constant, it is good practice to always include one. This prevents the compiler from ever treating a floating point constant as a fixed point constant, which could cause overflow.

String constants are specified by a series of up to 128 characters enclosed in apostrophes:

```
'Hello there'
'This is a string.'
```

To include an apostrophe in a string, use two consecutive apostrophes:

```
'If you can do that, that''s great'
'Don''t touch that dial!'
```

String constants are most often used to send messages to the user (with the PRINT statement) and to define literals (macros).

Variable Identifiers

An identifier is used to name a variable, procedure, or statement label. An identifier can be up to thirty-two characters in length, and is composed of letters, digits, or the characters number sign (#), dollar sign (\$), period (.), and underscore (_). The first character of an identifier cannot be a digit or a period. There is no distinction made between uppercase and lowercase letters. Some examples of identifiers are:

```
get user_input
CHAR2
array.length
TestCase
$element 23
f#ms_sector
```

There are a number of otherwise valid identifiers whose meanings are fixed in advance (but are not reserved). These are keywords such as IF, DO, and DECLARE. Because they are actually part of the XPL language, it is recommended that they not be used as programmer defined identifiers.

Variable Declarations

Each variable used in a Scientific XPL program must be declared in a declaration statement before it is used. The declaration defines the variable name and tells the compiler what data type to associate with it. The simplest form of a variable declaration is:

```
declare <identifier> <type>;
```

The identifier is followed by one of the four XPL data type keywords: FIXED, FLOATING, BOOLEAN, or POINTER. Variables of the same type can be grouped together in the same declaration statement by separating the variable names with commas and enclosing them in parentheses. Some sample XPL declarations are:

```
declare i          fixed;
declare (a, b, c, d) fixed;
declare (x, y, z)   floating;
declare (found, done) boolean;
declare ptr         pointer;
```

The reserve word DECLARE can optionally be shortened to DCL.

A separate declaration statement is not required for each and every declaration. Instead of writing the two declaration statements:

```
dcl chr    fixed;
dcl (x, y) floating;
```

both declarations can be written in a single declaration statement like this:

```
dcl chr fixed, (x, y) floating;
```

If a declaration statement contains more than one declaration element, each element is separated by a comma. The declaration elements appearing in a single statement are completely independent of each other; it is as if they had been declared in different statements. The only difference is that the keyword DECLARE has not been repeated.

It is good programming style to declare each variable on a separate line with a comment on that same line explaining the function of that variable in that program or procedure.

Literals

The LITERALLY declaration defines a macro for expansion as the program is being compiled. An identifier is declared to represent a character string, which is then substituted for each occurrence of the identifier in subsequent text. The form of the literal declaration is:

```
declare <identifier> literally <string>;
```

The keyword LITERALLY (which can be shortened to LIT) is followed by a string constant that is to be substituted for <identifier> wherever it appears in the program. The following program section illustrates the use of this literal feature:

```
dcl forever lit 'while (true)';  
dcl pi      lit '3.14159';  
  
do forever;  
  x = r*pi;  
  ...  
end;
```

In the above example, the first literal uses the XPL boolean value true to create a meaningful notation for specifying infinite loops. The second statement allows the word pi to be used in the program instead of the number 3.14159 to increase program readability.

A common use of literals is defining numeric constants that are fixed for one compilation, but may change from one compilation to the next. Often it is desirable to make a literal declaration for some quantity such as a data buffer size, a string length, or a program sampling interval. Then by changing one line at the top of the program, all occurrences of that quantity can be automatically changed throughout the program.

Section II - Terminal Input and Output

Scientific XPL has several statements that allow for the transfer of data to and from the computer terminal. The PRINT and SEND statements convert numeric quantities from an internal format to a sequence of printable characters for presentation on a terminal or transmission to a remote computer system. For entering information into a program while it is running, the INPUT statement is used to enter numeric information and LINPUT is used for textual information.

The PRINT and SEND Statements

The PRINT and SEND statements are powerful statements that enable the computer to produce output in a format that is easy to read. PRINT is used to present numerical information or textual material on the computer terminal. SEND is used to send this same information to a remote computer or to a printer. The formats for PRINT and SEND are identical.

The keyword PRINT (or SEND) is followed by a series of subfields separated by commas. Each subfield can be an arithmetic expression, a string constant, or an expression starting with one of the keywords CHARACTER, OCTAL, or STRING. Each subfield in the PRINT statement is printed on the terminal screen in order from left to right. If a comma appears at the end of the last subfield (just before the semicolon), then a carriage return/linefeed pair is not transmitted at the end of the output line. Otherwise, a carriage return/linefeed is performed at the end of the PRINT statement.

A PRINT statement subfield which is a fixed point arithmetic expression is printed on the terminal in a six character (including sign), zero-filled field. A floating point value is printed in a nine character field, including decimal point and sign. If the floating point value is larger than this format allows, asterisks (*) are printed instead.

For example, the following program:

```
dcl x floating;
dcl i fixed;

x = 3.14159;
i = 2;

print 'The value of pi is: ', x;
print 'The value of ', i, '*pi is ', i*x;
```

produces the following output:

```
The value of pi is: +3.141589
The value of 00002*pi is +6.283178
```

If a subfield in a PRINT or SEND statement contains an expression that includes the multiplication or division of two fixed point numbers, that operation will be performed in floating point to provide as much resolution as possible:

```
dcl (i, j) fixed;

print i/j; /* floating point format division */
```

This feature can be overwritten by using the INT function:

```
print int (i/j); /* fixed point division */
```

The PRINT statement includes a special feature to simplify its use with a video terminal. If CONTROL-S is typed while the computer is performing output to the terminal, the computer will halt temporarily until CONTROL-Q is typed. This freezes the video display so that the user can read it. CONTROL-S can also be used to halt the output to hard copy terminals temporarily to change the paper or make other adjustments. Note that this feature is disabled when interrupts are enabled.

Special PRINT Formats: CHARACTER, OCTAL, STRING

There are three special formats that can be used in conjunction with the PRINT or SEND statements: CHARACTER, OCTAL and STRING.

The CHARACTER format is used to print individual characters on the terminal. The keyword CHARACTER (which can be shortened to CHR) is followed by an arithmetic expression enclosed in parentheses. The result of the expression is written to the terminal directly. The ASCII character format is often used to write control information to terminals or printers or to send 8-bit bytes to a remote computer. Any 8-bit value (0-255) can be sent. The following program segment demonstrates the use of CHR:

```
dcl a.ff lit "'014'"; /* ASCII formfeed */
dcl a.can lit "'030'"; /* ASCII cancel */
dcl a.esc lit "'033'"; /* ASCII escape */
dcl a.gs lit "'035'"; /* ASCII group separator */

print chr (a.gs),; /* go into graphics mode */
print chr (a.esc), chr (a.ff),; /* clear graphics display */
print chr (a.can),; /* get out of graphics mode */
```


The OCTAL format is used when an octal (base 8) printout of a fixed point number is desired. The keyword OCTAL is followed by the arithmetic expression to print enclosed in parentheses. A six character zero-filled field is used for octal numbers. The following program demonstrates the use of OCTAL.

```
dcl (i, j) fixed;

i = 73;
j = 1125;

print octal (i), ' ', octal (j); /* print base 8 equivalents */
print 'Octal sum is ', octal (i + j); /* print sum in octal */
```

The STRING format is used to print the contents of a fixed point array which contains an XPL string (such as that returned by LINPUT). The keyword STRING is followed by the name of the string enclosed in parentheses. Observe how PRINT STRING is used in the following program.

```
dcl buf (64) fixed;      /* array for holding text string */

linput buf;              /* get the text string from terminal */
buf (0) = buf (0) - 1;   /* get rid of final carriage return */
print string (buf);      /* echo string back to terminal */
```

The INPUT Statement

The INPUT statement allows the user to type numeric quantities into an executing program. The keyword INPUT is followed by one or more variable names, separated by commas. When an INPUT statement is started, the computer will print a question mark on the terminal. The user should then type in the requested numbers, separated by commas (with no spaces). The following program demonstrates this:

```
dcl x floating;
dcl i fixed;

print 'Type in two numbers',; /* final comma so no CR/LF */
input x, i;
print 'The numbers you typed were: ', x, ' ', i;
```

When executed, this program presents the user with the following (underlined text is typed by user):

```
Type in two numbers? .2,500
The numbers you typed were: +.2000000 00500
```

Be sure to PRINT an intelligent prompt line before using the INPUT statement, so that the user will know what is being requested.

Special Use of INPUT

The INPUT statement is designed to be used for simple interactive applications. If a single carriage return is pressed in response to a request for numeric output, a value of zero will be returned and a format error will not occur. This is often convenient when user actions are required. The following program segment is an example of this:

```
dcl i fixed;

print 'Please insert the second diskette';
print 'and press RETURN when ready',; /* input prints a ? */
input i;
```

The LINPUT Statement

The LINPUT statement is used to input a string of characters from the terminal into a fixed point array. The keyword LINPUT is followed by the name of the array where the string should be stored. Only one array can be specified. The entire line of input (including any commas, spaces, and the final carriage return) will be stored in the array. Observe the following program:

```
dcl user_input (64) fixed;

print 'Start typing!';
linput user_input;
user_input(0) = user_input(0) - 1; /* get rid of CR at end */
print 'You typed: ', string(user_input);
```

A fixed point array that is used with LINPUT must always be declared to contain 65 elements (0-64). Up to 128 characters (including the final carriage return) can be typed in. Always be sure to print a message on the terminal using a PRINT statement before doing the LINPUT so that the user will know what information is being requested and when to type it in. The LINPUT statement does not print a prompt character (such as a question mark) on the terminal so that flexible input formats can be implemented.

Character strings are discussed in more detail in the section "Arrays and Pointers".

Notes on INPUT and LINPUT

During processing of both numeric input (INPUT) and string input (LINPUT), control characters can be used to strike out typing errors or to delete the input line. CONTROL-X is used to delete and restart the entire line, and DELETE or CONTROL-Z is used to erase the last character typed.

Parity information is masked off automatically by the LINPUT routine. Terminals that use odd, even, 1 or 0 parity can be used. The information stored in the user array will always represent a 7-bit ASCII code (0-127). LINPUT allows entering of both uppercase and lowercase letters. If the programmer wishes to ignore case when processing the line of input, he should explicitly check for commands typed in either uppercase or lowercase.

Make sure to always declare an array being used with LINPUT to have at least 65 elements. LINPUT assumes it can accept up to 128 characters, so if an array is used that is not large enough, memory might be overwritten.

Section III - Operators and Expressions

A well-formed expression consists of operands (variables, constants, or function calls) combined with the various arithmetic, logical, and relational operators, in accordance with simple algebraic notation. Examples are:

$a + b - c$	$x < y$	$d \bmod 3$
$a * 2 + c / d$	$c \geq 32$	$a \text{ xor } b$

If an arithmetic expression used in a Scientific XPL program contains only fixed point constants, the compiler will evaluate the expression at compile-time and use the resulting number in place of the expression. This reduces the amount of memory required by a compiled program and increases the speed at which the program executes.

Arithmetic Operators

Scientific XPL provides two unary arithmetic operators and seven binary arithmetic operators. The unary operators are:

-	unary minus (two's complement)
+	unary plus

Unary minus is used to negate a value. Unary plus has no effect when used, but is provided for consistency.

The seven binary arithmetic operators are listed below:

+	Addition
-	Subtraction
*	Multiplication
/	Division
mod	Modulus (remainder)
%	Fractional multiply
fdiv	Fractional divide

All seven of these operators are defined for fixed point operands, while only +, -, *, and / can be used with floating point operands.

Addition and subtraction are identical for both signed and unsigned fixed point variables as long as the result of the operation is within the range of interest (-32,768 to +32,767 for signed quantities, or 0 to 65,535 for unsigned quantities).

Fixed point multiplication (*) produces a signed 16-bit product from two signed fixed point operands. Overflow will result from the multiplication of two fixed point numbers if the product is less than (more negative than) -32,768 or greater than +32,767. Fixed point division (/) produces a signed 16-bit quotient and is valid for all signed fixed point operands. Remember that fixed point numbers are always integers and that the result of a fixed point division is technically the greatest integer part of the quotient (the fractional part being represented by the remainder).

The MOD operator computes the remainder of a fixed point division and is valid for all signed fixed point operands. The definition of a remainder indicates that the result of MOD will always be a positive number (or zero).

```
if (i mod 2) = 0 then ... /* check if I is an even number */
```

The fractional multiply (%) and fractional divide (FDIV) operators perform computations that are useful in some situations. They can be used to extend the range of fixed point variables to provide an effective 32-bit resolution. Fractional multiply first computes a signed 32-bit product by multiplying two signed operands (-32,768 to +32,767). Then this product is divided by the number 65,536 to obtain the 16-bit result (i.e., the result is the upper 16 bits of the 32-bit product). This is effectively equivalent to multiplying the first operand by a fraction that is the second operand divided by 65,536. Fractional multiply can be used in this way to scale a number by a fraction between -.500000 and +.4999847 with a resolution of 1/65536. In the following example fractional multiply is used to save the full 32 bits of the multiplication of two fixed point numbers:

```
dcl ms_word fixed; /* most significant word of product */
dcl ls_word fixed; /* least significant word of product */
dcl (i, j) fixed; /* operands */

ms_word = i % j; /* upper 16 bits of product */
ls_word = i * j; /* lower 16 bits of product */
```

The fractional divide operator is used to perform 32-bit divisions with fixed point variables. As with fractional multiply, the operands must be signed integers in the range -32,768 to +32,767. Also, to prevent overflow, the first operand must be less than one-half of the second operand. In a fractional divide, the first operand is multiplied by the number 65,536 (this is done by loading a different register) to form a signed 32-bit product. This product is then divided by the second operand to produce the 16-bit result. This is effectively equivalent to computing a fraction by dividing the first operand by the second, and then multiplying the fraction by 65,536. Fractional divide is often used to generate numbers that are used as one operand of the fractional multiply operator.

```
j = i % (4 fdiv 9); /* multiply I by 4/9 */
```

The *, /, MOD, %, and FDIV operators can be used with fixed point operands in Scientific XPL to achieve high speed computations involving signed integer quantities. Fixed point operations are performed using a single computer instruction which takes on the order of one microsecond to perform. If a Hardware Multiply/Divide unit is connected to the system, these functions will be performed using in-line computer instructions to achieve the highest possible throughput. A multiplication operation will require on the order of two microseconds to perform, while the other operations will require nearer ten microseconds since a sign correction must be performed. Unsigned divisions can be performed at higher rates (approaching two microseconds) by using the READ function and the WRITE statement.

If there is no Hardware Multiply/Divide unit in the system, an internal routine is used to perform each operation. These routines generally require 100 to 300 microseconds to compute the result.

Floating point variables are processed in Scientific XPL by using internal subroutines that perform the various operations. The actual amount of time it will take to perform a floating point operation will depend on the hardware configuration. Generally speaking floating point operations can be performed at a rate of 1000 to 5000 operations per second.

Floating point variables in the range of plus or minus one billion to one billionth and the number zero (0.0) can be added, subtracted, multiplied, or divided without causing arithmetic overflow. Accuracies of seven significant digits are obtained when using floating point variables. All internal products should be kept in the allowable floating point range of plus or minus 5.50 E-20 to 9.00E18.

Scientific XPL includes a special feature that provides extended precision for multiplications and divisions of fixed point operands. When two fixed point numbers are multiplied together, a 32-bit signed product is computed internally by the compiler. If the multiplication is immediately followed by division by a fixed point number, then the entire 32-bit signed product is divided by the third operand to form the desired quotient. This special feature is only activated if both the multiplication and division occur in the same program statement, as shown below:

```
del (i, j, k) fixed;  
i = i*j/k; /* multiply i*j, divide 32-bit product by k */
```

Note that there is a limitation in the current XPL compiler when computing equations that use constants, or rather, any equation that the compiler tries to precompute. If the above equation was written using constants (e.g., $1000 \cdot 250 / 469$), the compiler would precompute the expression using 16-bit (rather than 32-bit) arithmetic, and the result would be incorrect. To prevent the compiler from precomputing this expression, the second constant must be a variable (e.g., $1000 \cdot i / 469$ where i has been set to 250).

Here are some examples of the arithmetic operators:

```
dcl (i, j) fixed;
dcl (x, y) floating;

i = -j;
j = i + 256;
x = (i - 2) + y;
y = k*count;
x = total/sector_count;
i = count mod 8;
j = i % count;
i = i fdiv j;
```


Relational Operators

The Scientific XPL operators that perform arithmetic comparisons between numeric quantities are shown below.

<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
=	Equal to
~=	Not equal to

Each of these operators requires two operands that can be of any data type. The result of any of these operations is a boolean value true or false. The symbols \neq and $<>$ are equivalent to \neq .

If both operands are fixed point, then a fixed point comparison is performed in one computer instruction. This is done by subtracting the two operands and testing for a negative or positive result. This requires that the two numbers being compared must differ by no more than +32,767 or -32,768 if the correct relationship is to be determined. In general, numbers in the range of -16,384 to +16,383 may be compared using all relationals without overflow. When comparing numbers outside this range, the numbers should differ by no more than 32,767 if correct results are to be obtained. These limitations do not apply, however, to the = and \neq relational operators.

If either operand is floating point value, then a floating point comparison will be performed. In this case, each operand can have a value anywhere in the allowed floating point number range. Remember that floating point comparisons may take much longer than fixed point comparisons, depending on the system configuration. There are no overflow problems associated with floating point comparisons.

Relational operators are often used in the IF or DO WHILE statements:

```
if a = b then ...  
do while (a >= 256);
```

The result of a relational operation is the boolean constant true if the relation is true and the boolean constant false if the relation is false. The result of a relational operation can be used in an arithmetic expression:

```
i = i + (j = k);
```

which increments i by one if and only if j = k.

Unsigned Relational Operators

There are six relational operators that are designed to operate on unsigned fixed point quantities. These relational operators are useful for comparing pointers or subscripts for large data arrays. The symbols for these operators are listed below:

ilt	Is less than
ile	Is less than or equal to
igt	Is greater than
ige	Is greater than or equal to
ieq	Is equal to
ine	Is not equal to

Each of these operators requires two fixed point operands. The result of any of these operations is either true or false. The IEQ and INE functions are equivalent to the standard = and != relational operators and are included only for compatibility and standardization. The other relationals operate on unsigned 16-bit integers in the range of 0 to 65,535. These relational operators may be used with any fixed point variable without overflow.

```
if i igt j then ...  
do while (ptr ine null);
```

Logical Operators

Scientific XPL provides three boolean operators:

not	Logical negation
and	Logical and
or	Logical or

The NOT operator requires one operand, while AND and OR require two operands. These operators are most often used when two or more expressions are to be evaluated and then compared in one program statement:

```
if ((i ile ptr) and (y < 0)) then ...  
do while ((x < y) or (x > z) or (i = 0));  
if ((x > 0) and (not found)) then ...
```

In the first example, the IF expression will evaluate to true if and only if both the expressions combined with AND evaluate to true. In the second example, the expression will be true if one or more of the expressions combined with OR is true. The NOT operator in the third example is used together with a boolean variable to make a comparison. If FOUND is false in the above example, the expression (not found) will evaluate to true; if FOUND is false, the NOT operator negates this, making the final result true.

The XPL compiler will evaluate only as much of a logical expression as is necessary to determine whether the expression is true or false. For example, if any term of an AND expression is false, the entire expression is false. Likewise if any term of an OR expression is true, the entire expression is true. Because the compiler will stop evaluating an expression as soon as the result is known, it is inadvisable to embed a function call that must always occur within a logical expression unless it is the first term. The XPL compiler guarantees that logical expressions will be evaluated left to right.

For example, if the following expression appears in a program:

```
if ((index ile last) and ((not error) or (i <= 5))) then ...
```

and the first term (index ile last one) evaluates to false, evaluation of the whole expression will stop because it is clear that the final result will be false no matter what the other expressions evaluate to. This feature can be used to decrease program execution time if the order of the individual expressions in a complex expression is considered when the expression is written.

Bit Operators

There are seven operators in Scientific XPL that provide bit manipulation capability. These operators are:

not	One's complement
and	Bitwise and
or	Bitwise inclusive or
xor	Bitwise exclusive or
shr	Shift right
shl	Shift left
rot	Rotate left

The NOT operator requires one fixed point operand, while the others require two fixed point operands.

The NOT operator produces a result that is the one's complement of the operand. Each bit of the 16-bit operand is inverted to produce the corresponding bit in the result. For example:

```
not "000000" = "177777"
not "177760" = "000017"
```

The tilde (~) or the circumflex (^) can also be used to represent the NOT operator.

The operators AND, OR, and XOR compute a 16-bit fixed point result from two fixed point operands. Although the computer performs the function on all 16 bits simultaneously, each bit of the first operand is and'd, or'd, or exclusive or'd with the corresponding bit in the second operand to derive the result.

The ampersand (&) can be used to represent AND, while the vertical bar (|) or the backslash (\) can be used to represent OR. There is no alternate symbol for XOR.

The bit operators are used in arithmetic expressions in much the same way as their arithmetic counterparts. A single arithmetic expression may include both arithmetic and bitwise operators. For example:

```
i = (i and (not byte_flag)); /* turn off BYTE_FLAG */
j = (j xor 1);               /* toggle LSB of J */
```

The operators SHL, SHR, and ROT also perform bit manipulation on a 16-bit fixed point operand. Calls to these three functions take the following form:

```
i = shl (<value>, <bit_count>);
```

The keyword SHL, SHR, or ROT is followed by two expressions separated by a comma and enclosed in parentheses. Both arguments must evaluate to fixed point results. The returned value is equal to the first argument shifted left (SHL), shifted right (SHR), or rotated left (ROT) BIT COUNT bit positions. Bits shifted off the left end (SHL) or the right end (SHR) are lost. Bits shifted into the left end (SHR) or shifted into the right end (SHL) will be zeros.

In all cases the bit count must be between zero and fifteen (inclusive) for the result of the shift/rotate to be defined. Do not perform shifts with bit counts less than zero or greater than fifteen.

```
times = shl (times, 1);           /* multiply by 2 */
i = shl (sects, 8); /* words from sectors (multiply by 256) */
j = shr (words, 8); /* sectors from words (divide by 256) */
dev = shr (f#ms_sector, 8); /* get device from upper byte */
byte_swap = rot(value, 8); /* exchange upper & lower bytes */
```

ABLE computers incorporate hardware instructions that perform shift left, shift right, and rotate functions. These instructions operate on a 16-bit quantity, shifting one or eight bit positions per instruction in approximately one microsecond. If the bit count specified in a shift/rotate function is a constant expression, the compiler will compute the appropriate number of shift/rotate instructions and emit them in-line.

The truth tables for AND, OR, and XOR are shown below:

AND

Input Bit A	Input Bit B	A and B
0	0	0
0	1	0
1	0	0
1	1	1

OR

Input Bit A	Input Bit B	A and B
0	0	0
0	1	1
1	0	1
1	1	1

XOR

Input Bit A	Input Bit B	A and B
0	0	0
0	1	1
1	0	1
1	1	0

Examples of AND, OR, and XOR:

```
"000001" and "000000" = "000000"
"000003" and "000001" = "000001"
"012571" and "177777" = "012571"
"012571" and "177776" = "012570"
```

```
"000001" or "000000" = "000001"
"000003" or "000001" = "000003"
"012571" or "177777" = "177777"
"012571" or "177776" = "177777"
```

```
"000001" xor "000000" = "000001"
"000003" xor "000001" = "000002"
"012571" xor "177777" = "165206"
"012571" xor "177776" = "165207"
```

Assignment Statement

Results of computations are typically stored in variables. At any given moment, a variable has only one value, but this value can change during program execution. The assignment statement respecifies the value of a variable. Its form is:

`<variable> = <expression>;`

The expression to the right of the equals sign is evaluated, and the resulting value is assigned to the variable named on the left. The old value of the variable is lost. For example, following the execution of the statement

`a = 3;`

the variable A will have a new value of 3.

Numeric Conversions

While evaluating an expression, the XPL compiler is sometimes forced to convert an operand to another data type in order to carry out the operation. There is currently only one implicit conversion that is carried out by the compiler: if either operand of a binary operation is floating point, the other operand will be converted to floating point before the operation (addition, subtraction, etc.) is performed. No information is lost when this conversion is performed.

Likewise if the expression on the right hand side of an assignment statement yields a fixed point result (i.e., the expression involves only fixed point operands), and the receiving variable is of type floating, the fixed point result is converted to floating point before storage. If the result of an expression involving floating point operands is to be assigned to a fixed point variable, the INT function must be used to extract the integer value from the floating result. If the floating result that is passed to the INT function contains a fractional part, information will of course be lost in the conversion from floating to fixed point. Erroneous results will occur using INT if the value of the floating point expression is not in the range of -32,768 to +32,767.

```
dcl (i, j) fixed;
dcl (x, y) floating;
```

```
i = i*j;           /* fixed result assigned to fixed */
x = i + j;         /* fixed result assigned to floating */
y = x*i;           /* implicit conversion of I to floating */
i = int (x - y);    /* floating result assigned to fixed - note
                    that the INT function is required */
```


Precedence and Order of Evaluation

Operators in XPL have an implied precedence, which is used to determine the manner in which operators and operands are grouped together. The expression $A + B * C$ causes A to be added to the product of B and C . In this case, B is said to be bound to the operator $*$ rather than the operator $+$, and as a result, the multiplication will be performed first. In general, operands are bound to the adjacent operator of highest precedence, or to the one on the left in the case of a tie (XPL expressions are evaluated left to right). Valid XPL operators are listed below from highest to lowest precedence. Operators listed on the same line are of equal precedence. Equal precedence operations are evaluated left to right.

shr	shl	rot																		
not																				
*	/	mod	%	fdiv																
+	-																			
=	~=	<	<=	>	>=	ieq	ine	ilt	ile	igt	ige									
and	or	xor																		

Parentheses should be used to override the assumed precedence. Thus the expression $(A + B) * C$ will cause the sum of A and B to be multiplied by C . For example:

$a + b + c + d$	is equivalent to	$((a + b) + c) + d$
$a + b * c$		$a + (b * c)$
$a + b - c * d$		$(a + b) - (c * d)$
$a + (b - c) * d$		$(a + ((b - c) * d))$

When using the AND and OR operators in IF and DO WHILE statements, it is imperative that parentheses be used to clarify the desired relationship. For example, the statement:

if (a and 2) = 0 then ...

directs the computer to AND the variable A with the number 2 and test the result for zero. If it were written without parentheses, the computer would first test the variable A for a true or false condition (by testing A to see if it was even or odd) and then proceed to determine if the number 2 was equal to the number 0 (which of course it is not). Always use parentheses in such situations to avoid errors.

Section IV - Flow of Control

Scientific XPL has several constructs that are used to perform conditional branching, loops, and general control of program execution. By using the constructs described in this section, such as the IF, DO WHILE, and DO CASE statements, the programmer has very specific control over which program statements are executed when, and how many times those statements are executed.

Compound Statements

It is often desirable to group many program statements together and to treat that group as a single statement. This concept is most often used when creating IF or DO WHILE constructs. It is also used to logically group parts of a program together, thus organizing the program and making it easier to read and understand.

The most common way to form a compound statement is to bracket the program statements within the keywords DO and END. Generally the statements within a DO group are indented several spaces (usually three) to improve program readability.

```
do;  
    <statement>;  
    <statement>;  
    <statement>;  
end;
```

A bracketed group of statements is regarded as a single statement and may appear anywhere in a program that a single statement may. The DO-END group is most often used to group a number of conditionally executed statements together, such as when it is part of an IF statement.

Sometimes it is desirable to restrict the scope of variables to a small section of the program. In this case, the keyword BEGIN is used rather than the keyword DO.

```
begin;  
    <local declarations>;  
    <statement>;  
    <statement>;  
end;
```

Statements grouped together using BEGIN and END are considered a separate program block, in the same way that a procedure is considered a program block. Local variables can be declared inside a BEGIN-END block that have scope only within that block. This concept and how it is used will be discussed further in the section on variable scope.

IF Statement

The IF statement is used to selectively execute one or several program statements based on the result of a conditional expression. The basic form of the IF statement is:

```
if <expression> then <statement>;
```

The expression following the keyword IF is evaluated. If the result is true, then the statement is executed; if the result is false, nothing happens. Control then passes to the statement following the IF construct. Recall that in XPL any odd number evaluates to true, and any even number evaluates to false.

Some examples of the IF statement are:

```
if x < 0 then x = 0;
if (i = 2) and (count >= 10) then i = i + 1;
```

The IF statement can also be used with a compound statement, so that a group of statements are executed if the expression is true. In the next example, the three statements in the DO-END group would only be executed if the expression `temp < 100` were true:

```
if temp < 100 then do;
    value = value*2 + 1;
    count = count + 1;
    print 'Count = ', count;
end;
```

If a separate set of program statements should be executed if the condition is false, the ELSE clause should be used. In this case, either the THEN clause is executed (if the condition is true) or the ELSE clause is executed (if the condition is false).

```
if (x < 0) or (x > 100) then do;
    print 'Error - value out of range.';
    x = 0;
end;
else do;
    x = x*2 + 1;
    print 'New value = ', x;
end;
```

The programmer should use caution to avoid dangling ELSE clauses.
For example:

```
if i = 1 then if j = 2 then i = i + 1;
else j = j + 1;
```

The ELSE clause above goes with the second IF statement, that is, with IF j = 2, rather than with IF i = 1. To avoid this confusion, the program segment above could be written this way:

```
if i = 1 then do;
  if j = 2
  then i = i + 1;
  else j = j + 1;
end;
```

DO WHILE Loops

The DO WHILE statement is used to specify iterative loops, and takes the following form:

```
do while (<expression>);
  <statement>;
  <statement>;
  <statement>;
end;
```

In this construct, first the expression following the keyword WHILE is evaluated. If the resulting value is true, the statements within the DO-group are executed. When the END statement is reached, the expression is evaluated again, and the sequence of statements is executed again if the value of the expression is still true. The group is executed repeatedly until the expression evaluates to a false value, at which time execution of the statement group is skipped, and program control passes to the statement immediately following the END statement. Remember that in XPL true and false conditions are represented by odd and even values, respectively.

Consider the following example:

```
x = 0;

do while (x <= 4);
  x = x + 1;
end;
```

In the above example, the statement $x = x + 1$ will be executed exactly five times. When program control passes out of the loop, the value of x will be 5.

A useful form of the DO WHILE statement is as follows:

```
do while (true);  
    ...  
end;
```

The above example is an infinite loop. Many simple programs need to loop indefinitely performing a specific control function.

An interesting observation should be made at this point. The Scientific XPL compiler produces highly optimized object code, particularly in the area of conditional transfer instructions. Infinite DO WHILE groups such as DO WHILE (TRUE), DO WHILE (1=1), or even DO WHILE (2+2=7-3) produce absolutely no object code. Since the above expressions involve only constants, the compiler can evaluate the true and false condition as the program is being compiled. The XPL compiler will detect such constructs and simplify the object code accordingly.

Iterative DO Loops

An iterative DO loop executes a group of statements a fixed number of times. The simplest form of this construct is:

```
do <variable> = <expression1> to <expression2>;  
    <statement>;  
    <statement>;  
    <statement>;  
end;
```

where <variable> is a non-subscripted variable (the loop variable). The effect of this statement is first to store the value of expression1 into the loop variable. If the value is less than or equal to expression2, the DO-END group is executed. The loop variable value is then incremented by one, and the test is repeated. The DO-END group is repeatedly executed until the value of the loop variable is greater than expression2. At this point, the test fails, execution of the DO-END group is skipped, and control passes to the statement after the END statement.

Note that the value of expression2 is evaluated once before the loop is entered and never evaluated again. Furthermore, since an arithmetic signed comparison is used to test for the end of the loop, the difference between expression2 and expression1 cannot be greater than 32,767. Following the execution of a DO loop, the value of the loop variable is undefined.

Some iterative DO loops are similar in effect to DO WHILE loops. For example:

```
do i = 1 to 10;
  x = x + count;
end;
```

The above DO loop is equivalent to the following DO WHILE loop:

```
i = 1;
do while (i <= 10);
  x = x + count;
  i = i + 1;
end;
```

The only difference between these two types of loops is that if expression2 is a variable or other non-constant expression, it is evaluated every time through the DO WHILE loop, but only once (before the loop is entered) for the iterative DO loop.

To increment the loop variable by a value other than one, the optional BY clause can be added to the iterative DO loop:

```
do <variable> = <expression1> to <expression2> by <expression3>;
  <statement>;
  <statement>;
  <statement>;
end;
```

In this case, the loop variable is incremented by the value of expression3, instead of one, each time the END is reached.

```
/* compute the product of the first y odd integers */

del (x, y) floating; /* loop variable, limit */
del prod floating; /* product */

y = 4; /* initialize variables */
prod = 1;

do x = 1 to (2*y - 1) by 2;
  prod = prod*x;
end;
```

Negative loop increments are only allowed if expression3 is a constant. In the case of a negative loop increment, the loop control statement evaluates to true if expression1 is greater than or equal to expression2, as in this example:

```
do i = 100 to 25 by -5;
  total = total + i;
  print i;
end;
```

This loop will be executed for values of i at 100, 95, 90, ..., 30, and 25.

DO CASE Statement

The final form of selective execution is the DO CASE statement. The form of this statement is:

```
do case (<expression>);  
    <statement1>;  
    <statement2>;  
    ...  
    <statementN>;  
end;
```

The first thing that happens in the above form is the evaluation of the expression following the keyword CASE. The result of this is an integer value K which must lie between 0 and N-1. K is used to select one of the statements of the DO CASE group, which is then executed. The first case (statement1) corresponds to K=0, the second case (statement2) corresponds to K=1, and so forth. After one statement from the group has been selected and executed, control passes beyond the END statement of the DO CASE. If the runtime value of K is greater than the number of cases, then no case is executed and control passes beyond the END immediately. It is good programming practice to explicitly check the limits of the CASE expression before the DO CASE is executed.

Here is an example of the DO CASE statement:

```
do case (score);  
    ; /* case 0 */  
    conversions = conversions + 1; /* case 1: conversion */  
    safeties = safeties + 1; /* case 2: safety */  
    fieldgoals = fieldgoals + 1; /* case 3: field goal */  
    ; /* case 4 */  
    ; /* case 5 */  
    touchdowns = touchdowns + 1; /* case 6: touchdown */  
end;
```

When execution of this CASE statement begins, the variable SCORE must be in the range of 0 to 6. If SCORE is 0, 4, or 5 then a null statement (consisting of only a semicolon, and having no effect) is executed. Otherwise, the appropriate variable is incremented.

A more complex example of the DO CASE statement is:

```
do case (x - 5);  
  x = x - 5;          /* case 0 */  
  do;                /* case 1 */  
    x = x + 10;  
    y = y + x - 3;  
  end;  
  ;                  /* case 2 */  
  do i = 3 to 10;    /* case 3 */  
    a = a + i;  
  end;  
end;                  /* end of DO CASE */
```

The above example illustrates the use of DO-END blocks to group several statements as a single (although compound) XPL statement.

Statement Labels

Program statements can be labeled for identification and reference. A labeled statement takes the form:

```
<label>: <statement>;
```

where <label> is any valid, otherwise unused, identifier. Here are some examples of labeled statements:

```
LOOP: x = x + 1;  
INITIALIZE: i = 0;
```

It is also valid to have a label without a program statement following it, to be used as a target for control transfer. The customary semicolon is not needed in this case:

```
ERROR:  
PRINT.MESSAGE:
```

Note the occurrence of the colon (:) after the label in each case. The purpose of a label is to be a documentation and debugging aid, and to provide a target for GOTO statements.

The GOTO Statement

A GOTO statement stops the normally sequential order of program execution by transferring control directly to its target statement. Sequential execution then resumes, beginning with the target statement. The format for the GOTO statement is:

```
goto <label>;
```

where <label> is an identifier which appears as a label in a labeled statement (see above). The effect of the GOTO statement is a transfer of program control directly to the labeled statement. For example:

```
goto ERROR;
```

transfers control to the label ERROR defined above.

A discussion of label scope, which affects the legality of certain GOTO statements, is postponed to the section on scope.

A final note on labels: it is strongly recommended that the IF and DO statements be used instead of labels and GOTO statements wherever possible. The effect in general will be better object code and more readable programs. Only use a GOTO statement if there is no way to organize your program using the IF and DO statements to achieve the desired transfer of control.

Section V - Arrays and Pointers

Scientific XPL provides several data structures for grouping information in manageable ways. One-dimensional arrays allow for creating sets of numbers or characters, as well as the ability to access or change elements individually. Character strings have a special format in XPL, and the BYTE and PBYTE routines allow direct manipulation of string arrays. The DATA declaration defines static sets of numbers (or a string) that can be accessed in much the same way arrays are accessed. Finally, this section will cover pointers, and the direct manipulation of absolute memory locations.

Arrays

It is frequently convenient to let one identifier represent more than one value. Variables that have been declared to represent more than a single data element are called arrays or vectors. Arrays are often used to logically group data elements together, because all data elements can be referred to using the same variable name. XPL currently supports one-dimensional arrays only. A special form of the array structure is also used to store character strings, making character and string manipulation much simpler for the programmer.

The declaration statement for an array must contain the variable name associated with the array, the number of data elements in the array, and the type of the array elements. For example:

```
declare x (100) fixed;
```

This declaration statement causes the identifier X to be associated with 101 data elements, each of type fixed. The 101 data elements may be referred to individually with the names X(0), X(1), X(2), and so on up to X(100). The number in parentheses, which selects the specific data element of the array, is called a subscript or array index. Note that the lower bound of an array is always zero, while the upper bound is the declared size.

It is also possible to declare more than one array of a particular size and type within one declaration statement, such as in the following example:

```
declare (a, b, c) (199) floating;
```

This causes the three identifiers A, B, and C each to be associated with 200 floating point data elements, so that 600 elements of type floating have been declared in all.

The various elements of an array can be accessed and assigned individually by using subscripts as described above. If the third data element is to be added to the fourth, and the result stored in the fifth data element, we can write the assignment statement:

```
x (4) = x (2) + x (3);
```

Remember that the elements of an array are numbered starting with zero, so that X(2) is actually the third element of the array.

Much of the power of an array lies in the fact that its subscript need not be a numeric constant, but can be another variable, or in fact any valid XPL expression. The following program will sum the elements of the array NUMBERS:

```
declare numbers (10) fixed;
declare (sum, i)      fixed;

sum = 0;                                /* initialize variable */
do i = 0 to 10;                          /* loop through all elements */
    sum = sum + numbers (i); /* add each value to total sum */
end;

print 'The sum of the array = ', sum;
```

Subscripted variables are permitted anywhere XPL permits a simple variable with the one exception that it is not legal to use an array element as the loop variable of an iterative DO group.

The Scientific XPL Compiler does not give an error if the subscript of an array is out of bounds, so care must be taken to check subscript values to make sure they are in range before using them in a program. Accessing array elements that are out of bounds will cause random words of memory to be read or written.

Character Strings

Textual information can be stored in fixed point arrays. A special format of the array structure has been adopted for user convenience and consistency. The zeroeth element of such an array contains the number of bytes used in the array (i.e., the character length of the string). The character data begins at element one of the array, with each word representing two characters (an 8-bit byte is required to store each character). More information on how strings are stored is provided in the appendix "Internal Representations".

Character strings in this format are easily read from and written to the terminal using the LINPUT and PRINT statements. See the sections on these commands for the input and output of character strings.

Scientific XPL offers two built-in functions (BYTE and PBYTE) that assist in the processing of textual information. These functions operate on 8-bit characters stored in a fixed point array in the standard string format.

BYTE and PBYTE

The BYTE function operates on 8-bit data elements stored in a fixed point array. This is most often done when processing characters that were typed in from the terminal (LINPUT) or characters that are destined for the terminal (PRINT STRING). The BYTE function could also be used when data can be expressed in an 8-bit quantity (0-255) and the programmer desires to make more efficient use of memory space.

The BYTE function is used to access one 8-bit byte stored in a fixed point array. The keyword BYTE is followed by the name of the array and the byte number (starting with zero) enclosed in parentheses.

```
declare arr (64) fixed;
declare i      fixed;

linput arr;          /* get character string from terminal */
print byte (arr, 3); /* print the fourth character */

do i = 0 to 10;      /* print the first eleven characters */
  print byte (arr, i);
end;
```

Note that byte zero is stored in the lower half of element one of the array. ARR(0) always contains the character length of the string.

The PBYTE routine is used to store an 8-bit byte into a specific byte position of a fixed point array. It is used in conjunction with BYTE to process 8-bit numeric quantities. The keyword PBYTE is followed by the name of the array, the byte number (starting with zero), and the new value to store in the specified byte position. Only the appropriate lower or upper byte of the array element is altered.

```
dcl arr (8) fixed;
dcl i      fixed;

do i = 0 to 15;
  call pbyte (arr, i, a.sp); /* fill string w/spaces */
end;

call pbyte (arr, 7, a.p); /* write a P to byte number 7 */
```

The DATA Declaration

Many times it is convenient to reference a list of numeric constants in much the same way one references the elements of an array. XPL provides the DATA declaration so that sets of numbers can be created and then each number can be accessed by position as with array elements. The DATA specifier is used in a declaration statement to create a data list with an initial set of values:

```
declare nlist data (1, 3, 8, 14);
```

The numeric constants enclosed in parentheses are stored into the data list starting with element zero. In the above example, the fixed point numbers will be stored in the four element data list NLIST as follows:

```
nlist (0) = 1
nlist (1) = 3
nlist (2) = 8
nlist (3) = 14
```

The numbers stored in a data list can not be changed during program execution. Each element of the data list is treated as a numeric constant. Assigning new values or changing data list elements is not allowed.

Floating point data lists can also be created by using the attribute FLOATING in a DATA declaration:

```
dcl flist floating data (0, 3, .999, 10.100, .45);
```

A string constant can be assigned a variable name with the DATA declaration:

```
declare version_date data ('1 May 1987');
...
print 'Program version ', string (version_date);
```

The string is stored in XPL string format, with element zero of the array containing the length of the string, and each character being stored as an 8-bit unit. If there is an odd number of characters in the string, then the upper half of the last word in the array will be a zero. For a more detailed discussion of how bytes are packed into a fixed point array, see the appendix "Internal Representations".

Pointers

It is sometimes necessary in complex programming situations to manipulate memory locations directly. XPL provides functions that can be used to locate where variables or arrays are located in memory (ADDR and LOCATION), and also to directly read and write these locations (CORE). The data type POINTER is provided for declaring variables that are to be used as memory pointers.

The ADDR Function

The built-in function ADDR is used to find the absolute memory location of a variable or array element. This makes it possible to read data into the middle of an array or to an absolute location of memory.

The ADDR function is followed by the name of a variable enclosed in parentheses. If the specified variable is an array, then a subscript must be specified to select an array element:

```
del ptr pointer;

ptr = addr (count); /* location of the variable COUNT */
ptr = addr (buf (3)); /* location of element 4 of BUF */
ptr = addr (buf (i + 3));
```

ADDR returns a pointer to the absolute location of memory in which the specified variable is stored.

The CORE Array

Scientific XPL includes a special array called CORE that is used to read and write data from any location of memory. CORE is an array representing internal memory that starts at location zero of memory and has as many elements as there are words of internal memory. CORE is sometimes used in conjunction with ADDR to implement storage allocation routines. Obviously, the user must be careful when writing data into memory to use only locations that are not currently used by the program. More information on program memory can be found in the appendix "Memory Layout".

To read data from memory, CORE is used in an arithmetic expression with a subscript that indicates the location of memory to read. Information can be stored into memory by using CORE in an assignment statement.

```
print core (8192); /* print contents of location 8192 */
x = core (i);
core (8192) = 0; /* set location 8192 to zero */
core (ptr) = stack_top*3;
```

The specified location of memory is accessed as if it were a fixed point variable. If the program tries to read a location of memory that does not exist, the computer will halt and the program will stop running.

The LOCATION Function

The LOCATION function is used to reference absolute locations of memory during a procedure call. The keyword LOCATION (or the shortened form LOC) is followed by an expression, variable, or constant in parentheses. LOCATION can only appear in an actual parameter list during a procedure call. Observe the following example:

```
call readdata (0, 0, location (8192), 256);
```

When LOCATION is used, the computed expression is passed to the procedure as the start of an array. When using the LOCATION function the programmer must be sure to protect the locations of memory in which his program is residing. The memory map printed by the compiler in response to the COMPILE command can be used to determine this information, or refer to the appendix "Memory Layout" for information about determining free memory boundaries at runtime.

Interesting results can be obtained by using ADDR in conjunction with LOCATION. This is most often done when the user wishes to read data from the disk into element one of an array rather than element zero:

```
call readdata (0, 0, loc (addr (buf (1))), 256);
```

By using the LOCATION and ADDR functions, the user can implement dump and patch routines for referencing absolute locations of memory directly. Obviously, the user can only reference locations of memory that are unused by his program if system crashes are to be avoided.

Section VI - Procedures

The procedure definition facility of Scientific XPL allows modular programming, so that a program can be divided into logical sections. A well-designed procedure does a single application-related task (e.g., find the cosine of an angle, push an item onto the stack, or emulate a VT100 terminal). Furthermore, communication to that procedure should be by parameter passing only and the parameters should be restricted to application data (e.g., the angle to find the cosine of or the item to push onto the stack). Such procedures are conceptually simple, easy to debug and maintain, and easily incorporated into a large program. They can form a basis for a procedure library, if a family of similar programs is being developed.

A procedure is defined near the start of the program, and is then invoked (or called) from other parts of the program. During the procedure call, program control is transferred from the point of call to the top of the procedure, the procedure body is executed, and then program control is returned to the point of call.

A full discussion of XPL procedures is contained in this section, following an introductory section on XPL program structure.

Program Structure

Scientific XPL programs generally have a common structure, so that events happen in a certain order. One reason for this structure is the basic rules of the language, such as XPL variables and procedures must be declared before they are used. Another reason is that XPL programs should incorporate the elements of good programming style, such as the use of procedures, literals, generous comments, and general top-down design.

One of the most useful tools in programming is the ability to put comments into source code files. Comments can improve readability and provide important program documentation. An XPL comment is a sequence of characters that begin with the character pair `/*` and end with the character pair `*/`. These delimiters instruct the compiler to ignore any text between them. A comment can appear anywhere a space character may, thus comments can be freely distributed throughout an XPL program. XPL comments cannot be nested.

Because XPL procedures must be declared before they are used, an XPL program has the general structure of having the low-level procedures first, working to the top-level procedures near the end. The main program code is usually the last thing in the source code file. Global literals and declarations should appear at the beginning of the program, so that they can be used throughout the program and are easily found by the programmer. Block comments should be used to describe the function of each procedure, and individual comments should accompany each non-obvious program statement.

The basic guideline to follow is that an XPL program should be organized so that it is easy to read and understand. Procedures should be grouped according to function, and comments should explain clearly what each section of the program is doing. Declarations should be located where they would logically be looked for by another programmer. Such considerations make understanding and maintaining programs much faster and easier.

Notice the following outline of general XPL program structure.

```
/* Program Name

Information here in the comment telling about the program,
such as:
    - what the program does
    - general algorithms or methods
    - any special constraints or necessary hardware
    - acknowledgements, programmer's name
    - date last modified, who did it, what was done
*/

global literals
global declarations

/* Block comment about the following procedure */

Procedure definition:
    parameter declarations
    local procedure declarations

    procedure statement /* comment what it does */
    procedure statement /* comment this one too */
    ...
end procedure;

/* Block comment about the following procedure */

Procedure definition:
    parameter declarations
    local procedure declarations

    procedure code
end procedure;

/* Comment that says MAIN CODE */

main program declarations
main program body
```

Procedures

A procedure is part of a program that is separately defined, and is then invoked from other parts of the program. When a procedure is called, program control is transferred from the point of call to the top of the procedure, the procedure body is executed, and then program control is returned to the point of call.

During the procedure call, parameters (also called arguments) can be specified and passed to the procedure. Each parameter is either an expression or an array. Additionally, each procedure can return a value for use in an arithmetic expression. All these features will be described in the following sections.

A procedure must be defined before it can be used by a program. The basic procedure definition consists of the procedure name and the procedure code body:

```
<proc name>: procedure;
    <local variable declarations>

    <statement>;
    <statement>;
    ...
end <proc name>;
```

The procedure name is an identifier which will be used in later reference to the procedure. Any legal identifier can be used for the procedure name. In the procedure definition, the procedure name is followed by a colon (:) and the keyword PROCEDURE (which can be shortened to PROC). The statements within the procedure body can be any valid XPL statements, including definitions of other procedures.

A procedure is invoked with the CALL statement:

```
call <proc name>;
```

At the point of the CALL statement, program control passes to the procedure definition. The procedure body is executed, and when it is finished, control is returned to the statement immediately following the CALL statement.

Parameter Passing

Procedures can have one or more parameters. Parameters are values (or arrays) that are passed to the procedure at the point of call for use by the procedure body. For example, if a procedure is defined to output error messages, a special error code could be passed to the procedure when it is called and then the appropriate message would be printed on the terminal.

Parameters must be specified in two places: the CALL statement and the procedure definition. In order to pass parameters to a procedure, the CALL statement must include a parameter list, which is a comma-separated list of parameters enclosed in parentheses:

```
call <proc name> (<act1>, <act2>, ..., <actN>);
```

The parameters ACT1, ACT2, and ACTN are called actual parameters. They are the values actually passed to the procedure, and can be any valid expression or array. Different values or variables can be used as actual parameters each time a procedure is called.

In order for the procedure to properly receive the parameters it is being passed, the procedure definition must also include a parameter list. The parameters in the procedure definition are called the formal parameters, and take the following form:

```
<procname>: proc (<parm1>, <parm2>, ..., <parmN>);  
    dcl <parm1> <type1>;  
    dcl <parm2> <type2>;  
    ...  
    dcl <parmN> <typeN>;  
    <local variable declarations>  
  
    <statement>;  
    <statement>;  
    ...  
end <procname>;
```

Each of the formal parameters must appear in the parameter list, and each must be declared within the procedure body, preferably in the order in which they appear and before any other local declarations. When the procedure is called, the value of each of the actual parameters in the CALL statement is assigned to the corresponding formal parameter in the procedure definition. The data types of the actual and formal parameters must be the same.

Formal parameters are declared just as normal program variables with the exception of formal parameters that are arrays. In this case, the size of the array must not be specified, and the keyword ARRAY must be appended to the parameter's data type, as shown in the following example:

```
print numbers: proc (i, x, list);
  dcl i      fixed;
  dcl x      floating;
  dcl list fixed array; /* notice special form */

  print 'Integer      = ', i;
  print 'Floating point = ', x;
  print 'Array values  = ', list (0), ' ', list (1);
end print_numbers;

dcl a (9) fixed;
dcl y      floating;
...
call print_numbers (4, y + 1, a);
```

In the example, a procedure is used to print out four numbers in a particular format. Notice that the procedure is passed the correct number of parameters (three), and that each of them is the correct type (fixed, floating, fixed array). A compiler error will occur if the number and types of the parameters do not match.

When the procedure PRINT_NUMBERS is called, the formal parameters I, X, and LIST will take on the values specified in the actual parameter list. The parameter I will be 4, and X and LIST will become the values of Y + 1 and A at the time the procedure is called.

When an array is passed as a parameter, the size of the array is not specified in the DECLARE statement within the procedure body. Since arrays of any size can be passed as procedure parameters, the compiler does not check the limits of subscripts. Care must be taken not to exceed the maximum subscript of a passed array within a procedure body. If a programming error causes a subscript to exceed the amount of storage reserved for an array, random locations of memory will be accessed and perhaps overwritten. Extreme care should be taken to avoid this type of error.

Passing by Value

All scalar variables (i.e., everything except arrays) are passed to procedures by value. As the procedure is called, the current value of the specified variable is copied into a section of memory reserved for the corresponding formal parameter. If, within the procedure body, a new value is assigned to the formal parameter, the value of the actual parameter (the constant or variable that was specified in the CALL statement) will not be changed. For example:

```
put: proc (num);
    dcl num fixed;

    num = 30;
    print num;    /* NUM has been changed to 30 */
end put;

declare a fixed;

a = 10;          /* value of A is 10 */
print a;
call put (a);    /* procedure call... */
print a;         /* A is still 10 */
```

When the program is executed, the variable A is assigned a value of 10 and then printed on the terminal. The PUT procedure is called passing A as a parameter. At this instant, the value of A (which is 10) is copied into a special word of memory reserved for the formal parameter NUM. The PUT procedure is then initiated.

While PUT is being executed, a new value (30) is assigned to NUM. The variable A will not be affected by this, since different memory locations are used for A and NUM. The output of this program is:

```
00010
00030
00010
```

Passing by Reference

Arrays are passed to procedures by reference. Because each array may represent thousands of words of storage, it is impractical to duplicate each array element when a procedure is called. Instead, a pointer to the array is passed to the procedure. If, within the procedure body, a new value is assigned to an element of the array, that element will change in both the formal and actual parameters, since the same location of memory is used for both.

For example:

```
doit: procedure (a);
  decl a fixed array;

  a (5) = 25;      /* DOIT changes it to 25 */
  print a (5);
end doit;

declare list (10) fixed;

list (5) = 10;     /* this element is 10 */
print list (5);
call doit (list);  /* procedure call... */
print list (5);    /* and now it is still 25 */
```

The DOIT procedure requires one parameter, a fixed point array. When the fifth element of the formal parameter A is changed, the fifth element of the actual parameter LIST is also changed, since both arrays occupy the same storage area. The output of this program is:

```
00010
00025
00025
```


Type Conversion

The Scientific XPL compiler will automatically perform certain type conversions during a procedure call. No type conversion is required if both the actual and formal parameters are of the same type. If, however, a fixed point quantity is passed to a formal parameter of type floating, the specified quantity is converted to a floating point number before being stored. This conversion results in no loss of data or accuracy.

The compiler, however, cannot automatically convert from a floating point to fixed point quantity without risking the loss of important fractional information. If the user wishes to pass a floating point quantity to a procedure whose corresponding formal parameter is one of type fixed, the INT function must be used to convert the floating point value to fixed point. The fractional part of a floating point quantity is lost in the conversion from floating point to fixed point.

The following program demonstrates the different type conversions that can be performed by the compiler. POWER is a procedure that takes two parameters, the first of type floating and the second of type fixed. The procedure POWER raises the first parameter to the power of the second parameter and prints the floating point result.

```
power: proc (num, pow); /* raise a number to a certain power */
  dcl num    floating; /* number */
  dcl pow    fixed;    /* exponent */
  dcl result floating;
  dcl i      fixed;

  result = 1.0;          /* initialize RESULT */

  do i = 1 to pow;       /* loop POW times */
    result = result*num; /* raise NUM to the POW power */
  end;

  print result;          /* print the result */
end power;

dcl a fixed;
dcl b floating;

a = 4; b = 3.5;

call power (a, a);       /* fixed, fixed */
call power (a, int (b)); /* fixed, floating */
call power (b, a);       /* floating, fixed */
call power (b, int (b)); /* floating, floating */
```


In the first CALL statement above, POWER is passed two fixed point parameters. The first parameter (A) will be converted to floating point during the procedure call. In the second CALL, however, the INT function must be used to explicitly extract the integer part of the floating variable B for passage to the procedure POWER. Note that the parameter passing rules are identical to the assignment statement conversion rules.

The RETURN Statement

A procedure can return control to the point of call at any time by the use of a RETURN statement. Upon execution of a RETURN statement, a jump back to the point of call is performed, and any remaining lines of the procedure will not be executed. The RETURN statement is often used to check for certain conditions at the beginning of a procedure before it proceeds to execute the code body. For example:

```
print_message: proc (message);  
    dcI message fixed array; /* character string */  
  
    if message (0) = 0 then return; /* string is empty */  
  
    print; print 'Error! ',;  
    print string (message);  
    print;  
end print_message;
```

The above procedure should only print an error if it is passed a non-empty string. The length of the string is therefore checked at the beginning, and if the length is zero, the RETURN statement returns control back to the point of call without executing the rest of the procedure.

Never use a GOTO statement to leave a procedure body. Not only is this a poor programming technique, but information will continue to accumulate on the push down stack. Always use a RETURN statement to leave a procedure. Note that there is an implied return at the end of every procedure.

Functions

Functions are procedures that return a value back to the point of call. Functions can be very useful when computing arithmetic equations, or when error conditions or status codes need to be checked. The type of the value to be returned is specified with the RETURNS attribute of the procedure definition, which appears after the parameter list. The procedure type defines the precision of the value returned so that proper type conversion takes place when the procedure is invoked as part of an arithmetic expression. To return from a function, the RETURN statement followed by the value to return must be used.

A function can simply be called (to ignore the returned value), or a variable can be specified to receive the returned value. Functions can also be used within an expression. The following program is one example of how a function can be used:

```
sum: proc (a, b, c) returns (floating);
      dcl (a, b, c) floating;

      return (a + b + c); /* compute and return answer */
end sum;

dcl (x, y, z) floating; /* three input numbers */
dcl answer floating; /* result of the sum */

do while (true); /* loop forever */
  print 'Enter three numbers x,y,z',; /* get three numbers */
  input x, y, z;

  answer = sum (x, y, z); /* add them up */

  print 'Sum      = ', answer; /* print the sum */
  print 'Average = ', answer/3; /* print the average */
  print;
end;
```

The procedure SUM is a function that returns the sum of three numbers. Notice the use of the RETURN statement in the procedure body, and also the attribute RETURNS (FLOATING) in the procedure definition following the parameter list. A function must always end with a RETURN statement. Any RETURN statements within a function must specify the value that is to be returned.

This procedure definition could also be written without the word RETURNS in it, for compatibility with older programs:

```
sum: proc (a, b, c) floating;
```

If the value to be returned is a fixed point number, it is also possible to omit the return type completely. The compiler assumes that if a value is returned by a function and the type is not specified in the procedure definition, the returned value is fixed point. Neither of these older forms are recommended, however, because the RETURNS attribute immediately tells the programmer that a value is returned from that procedure.

A procedure that returns a value can be called as part of a standard arithmetic expression. For example, the variable ANSWER could be eliminated in the above program if the program statements were as follows:

```
print 'Sum      = ', sum (x, y, z);
print 'Average = ', sum (x, y, z)/3;
```

When a procedure is called in this manner, the numeric value returned by the procedure (using the RETURN statement) is used in the expression computation. Arithmetic expressions that call procedures may be used anywhere XPL allows numeric expressions. Multiple or nested procedure calls are also allowed within one expression.

Functions can be used to improve code legibility and detect errors. The following program returns a boolean status to the main program:

```
print_square: proc (x) returns (boolean);
    dcI x floating;

    if x < 0 then return (false);    /* negative number */

    print 'Square root = ', sqr (x); /* built-in function SQR */
    return (true);                  /* okay, did it */
end print_square;

dcI num floating;

do while (true);                    /* loop forever */
    print 'Number to find square root of',;
    input num;

    if not print_square (num) then do; /* if FALSE returned */
        print 'Sorry, cannot take the square root of negative';
        print 'numbers. Please try again.';
    end;
end;
```

The procedure PRINT_SQUARE first checks the parameter it received to make sure it is a positive number. If it is not, the RETURN statement is used to return a false value which makes the logical statement 'IF NOT' in the main program work correctly. Such logical relationships can often be used in XPL programming to increase program readability, or to signal that a procedure encountered an error.

Block Structure and Scope

XPL is a block-structured language. This means that a certain portion of a program, namely a block, can be written so there is no unwanted interaction between the block and its environment. This desirable situation stems from the concept of scope. Entities which are declared within a block are inaccessible to statements or declarations outside the block, and a block can shield itself from the influence of entities declared outside the block by suitable declarations inside the block. The use of the same identifiers for different objects, one inside a block and another outside the block, creates no problem.

For example, there are two blocks in the following program:

```
declare (a, b) fixed;

a = 3;

begin;
  declare c fixed;

  c = a - 17;
end;

b = a + 200;
```

The BEGIN-END group constitutes a block, as does the entire program. The scope of the variables A and B comprises the entire program, because they were declared in the outermost block. The scope of the variable C is within the BEGIN-END block only, because C was declared within that block. This means that the variables A and B may be used anywhere in the program, while the use of the variable C is restricted to the BEGIN-END block. A reference to C located outside the BEGIN-END block would result in an error message from the compiler; outside its scope, the variable C does not exist.

All identifiers are subject to scope. This includes names that represent variables, arrays, procedures, literals, labels.

How Scope is Defined

A block is a BEGIN-END block, any procedure body, or the entire program. Each block limits the scope of those identifiers declared within it; they will be unknown outside the block. Given an identifier, its scope is determined by finding the point of its declaration, and looking forward and backward in the program text ("outward" from the declaration) to find the innermost block containing the declaration. The exact scope of the identifier then begins with its declaration, and ends with the end of that block.

The scope of an identifier, so defined, can have "holes" in it. If the scope contains an inner block, and the inner block contains a declaration that redefines the same identifier, then the scope of that inner declaration creates an area in which the outer declaration is temporarily inoperative, masked by the inner declaration.

Notice the following example:

```
1  declare (a, b) fixed;
2
3  b = 0;
4
5  p: proc (a) returns (floating); /* procedure P */
6      decl a floating;
7
8      return (a*a + b);          /* returns A*A+B */
9  end p;
10
11 a = p (2);                     /* call P, store result in A */
12
13 begin;                          /* start of block */
14     declare p (10) floating; /* P is now an array */
15     declare i      fixed;
16
17     do i = 0 to 9;              /* loop */
18         p (i) = 500 + i;
19     end;
20
21     a = p (2);                  /* move element of P to variable A */
22 end;                            /* end of block */
```

Consider the scope of the variable I in the above program. I is declared at line 15; the innermost block encompassing this declaration is the BEGIN-END block comprising lines 13 through 22. Therefore, the scope of the variable I begins with its declaration at line 15, and ends with the end of the block at line 22.

The scope of the variable B begins with the declaration at line 1, and ends with the end of the program at line 22. That is to say, the scope of B is the entire program. The case of variable A is similar, since it is declared simultaneously with B, but there is an important difference. The procedure P, whose definition begins at line 5, contains the declaration of another variable A whose scope is the body of the procedure P, lines 5 through 9. So there are two distinct variables named A in this program, declared at two different block levels. The scope of the outer A fails to be continuous; it extends from line 1 to line 4, and from line 10 through line 22. It is interrupted by the scope of the inner A, which occupies lines 5 through 9. Thus the multiplication at line 8 uses the inner A (the formal parameter of procedure P), and the assignment statement at line 11 assigns an initial value to the outer A, the one declared at line 1.

The scope of B is not interrupted by any inner declaration in procedure P. That is why the reference to B at line 8, although within the procedure, is nonetheless a reference to the global B declared at line 1.

Let us now consider the scope of the procedure P. Its definition begins at line 5, and the innermost block encompassing this declaration is the entire program. The scope of procedure P is the entire program (from line 5 down) with one exception. Notice that the identifier P is declared again at line 14, this time not as a procedure, but as a ten element floating point array. As in the case of the identifier A, this double declaration presents no difficulty because the declaration at line 14 is contained within an inner block, in this case the BEGIN-END block encompassing lines 13 through 22. Without this inner declaration of the identifier P, the scope of the procedure P would be from line 5 down; with it, the scope of the procedure is only from line 5 through line 13.

The double declaration of P - once as a procedure and once as an array - has a curious consequence. The two statements at lines 11 and 21 ($A = P(2)$), although lexically identical, have completely different meanings. Line 21 falls within the scope of the array declaration at line 14, and thus sets the variable A equal to the third element of the array P (which the iteration of lines 17 through 19 has left equal to 502). Line 11, on the other hand, falls outside the scope of the array P, and within the scope of procedure P. Thus the assignment of line 11 invokes the procedure P with an actual parameter of 2; within the procedure body the inner variable A becomes equal to 2; the value $2*2 + B$, or 4, is returned as the value of the procedure call, and the outer A gets assigned the value of 4.

The Scope of Labels

It is not usually required to explicitly declare label names. The first use of an undeclared label is itself an implicit declaration of the label, and this implicit declaration governs the scope of the label according to the rules mentioned above. But there are times when a programmer must override these implicit declarations with his own explicit declarations.

The form of this type of declaration is:

```
declare <label name> label;
```

Such a declaration specifies that the label will be defined at the block level of the declaration. This explicit label declaration is necessary only if the implied declaration does not satisfy the needs of the programmer.

Here is an example that shows why the explicit declaration is sometimes required:

```
1  dcl x      fixed;
2  dcl EXIT label;
3
4  x = x + 1;          /* start of outer block */
5
6  begin;              /* start of inner block */
7      ...
8      goto EXIT;
9      ...
10 end;                /* end of inner block */
11
12 EXIT: stop;         /* exit is here */
```

In the above program, the obvious intention is to branch from the inner block (at line 8) to the outer block (at line 12). If line 2 (the label declaration) was not in the program, the label EXIT would first be declared (implicitly) at line 8 (GOTO EXIT). Since the label was implicitly declared at line 8, its scope would span the innermost block from line 8 through line 10. When the compiler encountered the END at line 10, it would try to limit the scope of the label EXIT, only to find that it has not been defined. The explicit declaration at line 2 serves to expand the scope of the label EXIT to encompass lines 2 through 12 (essentially the entire program).

The Use of Block Structure

Transfer of program control from one block nesting level to another should always be done by entering the block at its beginning and leaving at its end, or, for a procedure body, leaving by means of the RETURN statement.

For example, a GOTO statement which contrives to jump into the middle of a procedure body will leave the runtime push down stack in an undefined state, and continued execution of the program will produce unpredictable results. A procedure body must always be entered by means of a call to the procedure. A GOTO statement leaving a procedure body has similar trouble with the runtime stack, since it bypasses the orderly return mechanism. Because of this, it is illegal to write a GOTO statement inside a procedure that transfers control outside of the procedure. In general, GOTO statements from the middle of one block to the middle of another should be avoided at all costs. Programs can often be reorganized to remove the need for such GOTO statements.

It is recommended that within any given block all declarations be put at the beginning of the block, preceding executable statements. The scope of such identifiers can then be visualized as the extent of the entire block. This simplification also prevents an important class of programming errors: mistaken identification of the "innermost encompassing block".

Programmers find their work greatly facilitated by proper layout of a program on the pages of its program listing. Blocks (procedures, DO-groups) are frequently set off by blank lines. The body of each block is indented by a fixed number of spaces (usually three) from the code in which it occurs, thus the opening and closing lines of the block are vertically aligned. When you look at a program listing it should be easy to see the block nesting structure at a glance, without reading the code in detail.

Block structure in a programming language provides the opportunity to define truly independent program modules, letting the compiler do the work of keeping them independent. Procedures can be made independent of their environment (except for the number and type of parameters). Procedures can be moved from one program to another, with no surprise results from the new declaration. Complete self-contained modules, together with conventional literal definitions, can form a project or department library, greatly reducing program development time.

Variable Storage Classes

The storage class of an XPL variable determines its lifetime (when it exists) and its scope (area of influence). There are four storage classes of variables, two of which are meaningful only within the context of procedures: STATIC and AUTOMATIC. Both of these types follow the scope rules explained in the previous section; they differ only in their lifetimes. The other two storage classes (PUBLIC and EXTERNAL) are explained in the next section, "Modules and Libraries".

Static variables exist over the entire lifetime of the program, from the moment the program starts executing until the moment it terminates. They are initialized (to zero) only once, at the start of program execution. Any value that is stored in a static variable is retained until explicitly overwritten by the programmer's code.

Automatic variables, on the other hand, exist only when the block in which they are defined is active (i.e., while that block is being executed). They are created and initialized (to zero) at the start of the block and destroyed when the block is finished executing. Thus they are dynamic, existing only when needed. They can be used to save memory space in a program, although the execution time of the program will necessarily be increased. Automatic variables can be used only in procedures.

The storage class of a variable appears directly after the variable type in the declaration statement. The following examples are all valid variable declarations:

```
    dcl i      fixed static;
    dcl x      floating automatic;
    dcl a (9)  boolean automatic;
```

All variables in XPL are static by default (i.e., if no storage class is specified). If a variable must be static in order for the program to work correctly, it is a good idea to explicitly declare it to be static for documentation purposes.

Automatic variables are allocated off the runtime push down stack. It is important to make sure this stack is large enough to avoid a stack overflow, or else erroneous program results will occur. If the default stack is not large enough, its size can be increased with the PDL statement (see the appendix "Compilation Control").

There is one restriction with the use of automatic variables that does not exist for static variables. Automatic variables cannot be accessed by a procedure that is nested within the procedure that declared the variable. If the nested procedure needs to access the variable, the variable must either be static or it must be passed to the procedure as a parameter.

Recursive Procedures

Variables in procedures are static by default. If a procedure's function is recursive in nature, it is possible to make the variables within that procedure default to automatic. This is done in the procedure definition by affixing the keyword `RECURSIVE` after the returns type:

```
<proc name>: proc (<parm list>) returns (<type>) recursive;
```

The following example is a recursive factorial function:

```
factorial: proc (x) returns (fixed) recursive;
  dcl x fixed; /* automatic by default */

  if x <= 1 /* 0! and 1! are both 1 */
  then return (1);
  else return (x*factorial (x - 1)); /* x*(x - 1)! */
end factorial;
```

Any local variable declared in a recursive procedure that needs to be static must explicitly be declared to be static. Note that recursive procedure formal parameters cannot be declared static; they must be automatic. If a procedure is defined within a recursive procedure, it is recursive by default and cannot be changed.

Recursive procedures quickly use up the runtime stack. See the warning about automatic variables above.

Swapping Procedures

For very large programs, it is possible to store certain procedures in external memory, and to swap them into internal memory when they are needed by the program. Procedures that can swap are identified by the keyword SWAP in the procedure definition:

```
<proc name>: proc (<parm list>) returns (<type>) swap;
```

SWAP must always appear last in the list of procedure options (RETURNS, RECURSIVE, then SWAP).

The XPL runtime system keeps at most one swapping procedure in internal memory at a time. If a procedure that swaps calls another procedure that swaps, the original procedure will be read back into internal memory after the called procedure returns. Thus, swapping procedures can potentially execute many times slower than procedures that do not swap. It is important to make sure that frequently called procedures are not swapping procedures.

The compiler includes special logic to detect string constants in swapping procedures that are not eventually passed to another swapping procedure. It also detects data arrays that are not passed as actual parameters to other procedures. Such string constants and data arrays are stored in external memory with the swapping procedure they are contained within, and swap into internal memory with that procedure when it is called. This feature saves a great deal of internal memory in large programs.

To minimize the amount of internal memory used by a program, all procedures that are not of a time critical nature should be defined to be both recursive and swapping. This causes a program to use only as much internal memory as it really needs at any given moment.

Programs that have been compiled with swapping procedures cannot run on systems that do not have external memory.

Forward Reference Procedure Declarations

It is possible to invoke a procedure that is defined later on in the program. This feature is used so that procedures can be grouped logically within a program rather than in the order in which they are used. The declaration statement for a procedure must occur before any references to the procedure, but the actual procedure definition can exist later in the program. The syntax allows any type of procedure to be forward referenced:

```
decl <proc name> proc (<parm list>) returns (<type>) <class>;
```

where <proc name> is the name of the procedure, <parm list> is a list of the parameter types separated by commas, <type> is the type of the value the procedure returns, and class is RECURSIVE if the procedure is recursive. If no parameters are passed, the <parm list> should be omitted. If no value is returned, the RETURNS attribute should be omitted.

If a procedure is recursive, the forward reference declaration must include the keyword RECURSIVE. However, if a procedure is a swapping procedure, the keyword SWAP cannot appear in the forward reference declaration.

Some examples of valid forward reference procedure declarations are:

```
decl x proc;
decl p procedure;      /* no SWAP attribute here */
decl r proc recursive; /* RECURSIVE attribute must be here */
decl y proc (fixed);
decl z proc (fixed, floating, fixed array) returns (floating);

p: proc swap;
  print 'Program name and date';
end p;

r: proc (i) recursive;
  decl i fixed;
  ...
end r;

z: proc (temp, div, values) returns (floating);
  decl temp    fixed;
  decl div      floating;
  decl values   fixed array;
  decl result   floating;
  ...
end z;
```

Section VII - Modules and Libraries

Scientific XPL provides several ways in which to divide programs into separate sections, as with the use of procedures and blocks. Another way to modularize a program is to place the program code in several different source files instead of one large file and to draw the code together at compile-time using the INSERT and ENTER statements. Different parts of a program can even be compiled separately, and then linked into the main program at a later time. Such program sections are called modules, and after they have been compiled they are referred to as libraries.

There are many reasons why dividing a program into separate files or modules is a useful programming tool. First of all, as programs become larger and larger, it becomes desirable to divide them into several independent sections, making the editing process easier. Secondly, a common library of procedures can be developed that can then be included in several main programs. For example, one can write a procedure to perform a special numeric calculation, store that procedure in a separate file, and then include it in any main program with proper use of the INSERT or LIBRARY statement. Finally, different sections of the program can be compiled and tested individually, simplifying the debugging phase of program development and reducing the time needed to compile the main program.

Treenames

The topics discussed in this section require that the programmer understand treenames. Treenames are used so that files in any catalog or on any device in the system can be accessed from the current catalog. They take the same format when used with the statements in this section as they do with Monitor commands, such as OLD and ENTER. When a single filename is specified, that file is searched for in the current catalog, and if it is not found, the system catalog is searched. If the file is located elsewhere, a pathname must be specified before the filename. A pathname is comprised of an optional device specifier and then one or more catalog names, separated by colons. A pathname with a filename appended is the complete treename for that file.

A leading colon is used to indicate the top level of the current device. If the colon is used with no subcatalog name, as in ENTER ':';, the top level catalog of the current device will be accessed. If the colon precedes a subcatalog name, as in ENTER ':SUBCAT1';, the specified subcatalog located in the top level catalog will be accessed.

The following are all valid treenames:

```
'a'          /* file A in current catalog */
':a'         /* file A from top-level catalog */
':test:a'    /* file A from catalog TEST in top-level */
'x:p:a'      /* file A from catalog P in cat X in current cat */
'w1:st:a'    /* file A from catalog ST on W1 */
'f0:a'       /* file A from F0 floppy disk */
```

The INSERT Statement

The XPL INSERT statement is used to include lines of program statements that are stored in a separate file. The INSERT statement allows full treenames to be specified:

```
insert '<treename>';
```

When the Scientific XPL Compiler processes an INSERT statement, the system is searched for the file specified. If the file is not found, a compiler error will occur. The inserted file may or may not be line-numbered. The body of textual material in the specified file is compiled into the final object code, exactly as if it were typed into the current source file at the location of the INSERT statement.

INSERT statements can be nested. The treename or filename is enclosed in apostrophes and appears after the keyword INSERT.

The following examples are all valid INSERT statements:

```
insert 'x';           /* insert file X from current catalog */
insert ':x';          /* insert file X from top-level catalog */
insert 'abc:hh:x';    /* insert file X from catalog ABC:HH */
insert ':abc:x';      /* insert file X from catalog :ABC */
insert 'f0:x';        /* insert file X from floppy disk in F0: */
```

Optimization of Procedure Source Libraries

The Scientific XPL Compiler includes an interesting optimization feature that simplifies the use of procedure source libraries. Many times, the programmer wishes to assemble a library of common procedures that are used in more than one program. When the compiler processes procedure definitions, it will specifically exclude from compilation any procedures that are not actually called by the program being compiled. By taking advantage of this feature, a programmer can construct a general purpose procedure source library that contains many common subroutines, knowing that only those procedures required by a particular program will actually be compiled.

The ENTER Statement

The ENTER statement is used in conjunction with the INSERT statement to include source files from different devices and subcatalogs into the program currently being compiled.

The ENTER statement changes the catalog that the compiler is currently accessing for inserted files. Without the use of an ENTER statement, all files are searched for starting from the catalog where the COMPILE command was issued (the current catalog). By using ENTER to change this default catalog, any source files that are then inserted will be searched for from the catalog specified in the ENTER statement. Note that the XPL statement ENTER works somewhat differently from the Monitor command ENTER, in that it does not actually change the current catalog.

The syntax for ENTER is similar to that of the INSERT statement:

```
enter '<pathname>';
```

The following examples are all valid ENTER statements:

```
enter 'ab';           /* access files from subcatalog AB */
enter ':ab:hi:c';     /* access files from subcatalog :AB:HI:C */
enter ':';            /* access files from the top-level catalog */
enter 'f0: ';         /* access files from the floppy disk in F0: */
```

The asterisk can be used to indicate a return to the current catalog, as follows:

```
enter '*';
```

where the current catalog is the catalog from which the command COMPILE or RUN was issued.

Here is an example of ENTER used with INSERT:

```
insert 'source';      /* inserts SOURCE from current catalog */
enter 'subcat';       /* changes access to subcatalog SUBCAT */
insert 'source2';     /* inserts SOURCE2 from SUBCAT */
```


Using ENTER with INSERT

The ENTER statement causes the specified catalog to be searched for and opened before succeeding files are inserted. This means the full treename does not need to be searched for every INSERT from that catalog. So, in the example:

```
enter ':abc:def:ghi';
insert 'x';
insert 'y';
insert 'z';
```

the catalog :ABC:DEF:GHI is found during the ENTER statement and then remembered for all the following INSERT statements. Contrast this with:

```
insert ':abc:def:ghi:x';
insert ':abc:def:ghi:y';
insert ':abc:def:ghi:z';
```

In the above example, it would seem that the pathname :ABC:DEF:GHI is searched for each INSERT statement. However, the INSERT processing is optimized so that after a pathname has been searched, each level of that pathname (up to some maximum) is remembered until a different pathname is specified. The pathname is therefore only searched once and the results are identical (with one exception) to using ENTER.

For example, the sequence:

```
insert ':abc:def:ghi:jkl:x'; /* search for :ABC:DEF:GHI:JKL */
insert ':abc:d';             /* we know where :ABC is */
insert ':abc:def:ghi:j';     /* we know where :ABC:DEF:GHI is */
insert ':abc:def:ghi:jkl:y'; /* this is the original pathname */
```

would also only cause one pathname search. However, the sequence:

```
insert ':abc:def:ghi:jkl:x'; /* search for :ABC:DEF:GHI:JKL */
insert 'xyz:d';              /* search for XYZ in current cat */
insert ':abc:d';             /* search for :ABC */
insert ':abc:def:ghi:p';     /* search for DEF:GHI in :ABC */
insert ':abc:def:ghi:jkl:y'; /* seek JKL in :ABC:DEF:GHI */
```

causes a variety of different searches.

The main difference between INSERT with a treename and ENTER has to do with what happens when an inserted file has its own INSERTs. What if file X (in the first example) has the statement INSERT 'a'; buried in it? Where do we search for file A in each case?

In the case of ENTER (the first example), the compiler will search for A in the subcatalog :ABC:DEF:GHI. In the second example (INSERT with a treename), the compiler will search for A in the last catalog entered. There is a tremendous difference.

Modules

It is often advantageous to split a large program into much smaller pieces, each of which can be compiled separately and only brought together (linked) when the final program is compiled. These smaller pieces are called modules.

The obvious advantage to this approach is that only the parts of a program that change need to be recompiled to create a new version of a program. For a large program under active development, the time savings can be immense.

The more important advantages lie in the fundamental concepts of structured programming. A well written program (i.e., one that is reliable, flexible, efficient, and maintainable) depends on the ability to partition a large problem into smaller, more manageable parts. The parts generally take the form of black boxes. In other words, their inputs are known, their outputs are known, their function (i.e., what they do to their inputs to create their outputs) is known, but how they do their function is not known or at least it does not need to be known. A properly designed black box can be implemented in many different ways without affecting the user of the box; the box's interface and function do not need to change when it changes. An integrated circuit is a good example of a black box.

XPL procedures provide a very simple black box mechanism. To use a procedure, it is only necessary to know its parameters (inputs), return value (output), and what it does (function). It is totally irrelevant how it does it.

Unfortunately, many black boxes provide multiple functions for the same well defined object. For example, a tape deck allows you to "stop", "play", "rewind", "fast forward", or "record" a tape. A black box that is suitable for managing stacks would allow you to "push" an item onto the stack, "pop" an item off the stack, check if the stack is "empty" or "full", and "clear" all items from the stack (force it empty). These functions all operate on a stack. XPL procedures in and of themselves do not provide this kind of mechanism. There must be some way to group related procedures together with their data structures.

Thus, XPL provides a more generalized black box facility: the module. A module is a region of code which is to be treated as an independent set of procedures which will later be linked with other procedures and some main body of code (referred to by the compiler as MAIN). It is defined using the MODULE statement.

A file which contains a module can contain nothing else, including other modules. Furthermore, the first XPL statement (note that comments are not statements) in that file must be a MODULE statement of the form:

```
module <module name>;
```

and the last statement in the file must be an END statement of the form:

```
end <module name>;
```

where the module name in the END statement is optional, but must match the module name in the MODULE statement if it appears. Any statements found outside this module are disallowed. Nesting of modules is also disallowed. Between the MODULE statement and the END statement, any valid XPL statements can occur.

A word of caution about WHEN statements in modules: only one WHEN statement for any given condition can occur in the final linked program. For example, if WHEN D03INT occurs in the main body of a program, it must not appear in the source module of any referenced library. Furthermore, if a module is compiled for a Model C processor, a program that will run on a Model B processor cannot use that compilation of that module.

Public Variables and Procedures

Variables and procedures declared within a module are, by default, local to that module. That is, they cannot be accessed from any of the other modules in the program nor from MAIN.

Any variable or procedure, however, can have its scope extended to the entire program (i.e., made global) with the PUBLIC storage class. Like the storage classes STATIC and AUTOMATIC, PUBLIC must appear directly after the variable type in the variable declaration. The following are examples of public variable declarations:

```
dcl i      fixed public;
dcl x      floating public;
dcl a (9)  fixed public;
dcl l      label public;
dcl d      data public (0, 1, 2, 3, 4, 5);

p: proc (x, y) returns (fixed) public swap;
    dcl (x, y) fixed;

    return (x + y);
end p;
```

In the case of recursive or swapping procedures, PUBLIC must appear first, followed by RECURSIVE and then SWAP.

A public variable should not be declared inside a procedure unless it is certain that the procedure in question will be called. Although the compiler does not enforce this restriction, it does print a warning message to this effect.

Public variables are a special case of static variables. Like static variables, they are initialized (to zero) once at the start of program execution and exist for the life of the program. Their scope, also like static variables, is from the point of declaration to the end of the block in which they are defined. They differ from normal static variables in that they can be referenced (and hence accessed) from anywhere in the program through the use of the EXTERNAL storage class (see below). Thus, their scope is really the entire program. A given variable identifier can be declared to be PUBLIC only once in a program.

The module designer and programmer are encouraged to restrict the number of procedures and especially variables that are accessible from outside a module (i.e., public). A well designed module performs a defined task (e.g., stack manipulation) on localized data (e.g., a stack). Any access to this data is conducted through global functions (e.g., push, pop, empty, full, clear) rather than through global variables. Any internal partitioning of those functions as well as the internal representation of the data is invisible outside the module. A well designed module is, for all intents and purposes, a black box. The advantages to this approach are information hiding (the user doesn't need to know the internal details), data integrity (only the module's functions operate on the data), implementation independence (the implementation can be upgraded to be more efficient or provide additional capabilities without affecting code that accesses the module), and easy maintenance of both the module and any programs that use it.

External Variables and Procedures

In order to reference a public variable or procedure from a part of the program that is not in the scope of the original public declaration (e.g., from another module), it is necessary to redeclare the variable with the EXTERNAL storage class. This storage class is unique in that there is no storage associated with a variable declared in this way. Rather, it references a variable that has already been declared to be public elsewhere in the program.

Being a storage class, the keyword EXTERNAL must appear directly after the variable type in the variable declaration. In the case of procedures, an external declaration takes the form of a forward reference procedure declaration with EXTERNAL appearing at the end (RECURSIVE and SWAP are unnecessary and cannot appear). The following examples of EXTERNAL declarations correspond to the PUBLIC declarations above:

```
dcl i fixed external;  
dcl x floating external;  
dcl a fixed array external;  
dcl l label external;  
dcl d data external;  
dcl p proc (fixed, fixed) returns (fixed) external;
```

Each variable which is declared with the EXTERNAL storage class must have a corresponding declaration with the PUBLIC storage class somewhere in the program. Any variable that is declared to be external, but does not have a corresponding PUBLIC declaration is called an "unresolved reference" and is not allowed.

A variable can be declared with the EXTERNAL storage class many times throughout a program. It is good style, however, to declare an external variable only in modules where it is actually used. In this way, access to public variables can be restricted.

Libraries

When a module is compiled, a relocatable binary (or library) is created. The LIBRARY statement references a relocatable binary that needs to be "inserted" into the program, much as the INSERT statement references a source file that needs to be inserted into the program. The LIBRARY statement can appear anywhere an INSERT statement can appear and shares the same syntax:

```
library '<treename>';
```

The file specified must be a relocatable binary (a compiled module of file type RELOC).

Libraries are always linked in before the main body of code (MAIN) and so any executable statements that lie outside procedure bodies in libraries (e.g., local or global variable initialization) will be executed before the main body of code. The libraries will appear in the final program in the order in which the compiler finds the LIBRARY statements. Therefore, if only the main body of code has LIBRARY statements, the corresponding libraries will be linked in order. But if the first library inserted in the main body of code has LIBRARY statements within it, all those libraries will be linked in before the next main code LIBRARY statement is processed.

This information is important to know to guarantee that any executable statements located outside procedure bodies in the original modules are executed in the proper order in the final program. A list of the libraries linked in and the order in which they appear in the final program can be obtained by setting the appropriate compile-time option (see the appendix "Compilation Control").

For example, if the main body of code has the following statements:

```
library 'output';
library 'set';

dcl output proc external;
dcl set proc (fixed) external;

call set (3);
call output;
```

The source module for library OUTPUT contains:

```
module output;                /* output the contents of X */
  output: proc public;
    dcl x fixed external;
    print x;
  end output;

  call output;
end output;
```


The source module for library SET contains:

```
module set;                /* set X to some value */
  library 'init';
  dcl init proc external;

  call init;

  set: proc (new 'x) public;
    dcl new 'x fixed;
    dcl x      fixed external;

    x = new 'x;
  end set;
end set;
```

And the source module for library INIT contains:

```
module init;              /* initialize X */
  dcl x fixed public;
  x = 5;

  init: proc public;
    x = -1;
  end init;
end init;
```

The order of the LIBRARY statements in the main body of code determines the order of the libraries in the final program and hence the order of the executable statements that lie outside procedure bodies in those libraries. In the example above, the final position of each library in the final program would be as follows: OUTPUT, SET, INIT, and then MAIN. Therefore, the final program will execute the following statements in exactly this order:

```
call output; /* from OUTPUT - prints a 0 */
call init;   /* from SET    - sets X to -1 */
x = 5;       /* from INIT   - sets X to 5 */
call set (3); /* from MAIN  - sets X to 3 */
call output; /* from MAIN  - prints a 3 */
```

The intended order was presumably:

```
x = 5;       /* from INIT   - sets X to 5 */
call init;   /* from SET    - sets X to -1 */
call output; /* from OUTPUT - prints a -1 */
call set (3); /* from MAIN  - sets X to 3 */
call output; /* from MAIN  - prints a 3 */
```

To achieve this order, INIT must be libaried in the main body of code and must precede the LIBRARY statement for SET which must precede the LIBRARY statement for OUTPUT as follows:

```
library 'init';
library 'set';
library 'output';
```

Libraries that are referenced from modules do not have to actually exist until the linking process commences (which happens when you compile the main program). This feature is useful when designing a system with a top-down approach and you want to compile a module (or make sure a module compiles) that references a low-level library that perhaps has not been written yet.

Using ENTER with LIBRARY

The ENTER statement behaves somewhat differently when used with the LIBRARY statement than with the INSERT statement. Above, it was mentioned that libraries referenced from within modules do not have to exist until the main body of code is compiled. That is only true for the LIBRARY statement. If the ENTER statement is used to access a library, the catalog specified in the ENTER statement must exist at compile-time. For example:

```
module x;          /* define module X */
  library 'abc:x:z'; /* ABC:X:Z need not exist at compile */
  enter ':def:xx';  /* :DEF:XX must exist at compile-time */
  insert 'ccc';     /* CCC must exist at compile-time */
  library 'ddd';    /* DDD need not exist at compile-time */
end x;             /* end of module X */
```

Recall that when the ENTER statement is used in conjunction with the INSERT statement, any files which are inserted from the inserted file are searched for in the catalog entered at the time of the initial INSERT. For example, if file A contains:

```
enter 'xyz';
insert 'b';
```

and file B contains:

```
insert 'c';
```

then file C is searched for in catalog XYZ instead of the current catalog. This is not true when the ENTER statement is used in conjunction with the LIBRARY statement. When a library is inserted, all searches initiated from within that library commence from the current catalog.

For example if file A is the main body of code and contains:

```
enter 'xyz';  
library 'b';  
library 'd';
```

and file B's source module (remember B is a precompiled module) contains:

```
module p;  
  library 'c';  
end p;
```

then file B and file D are searched for in subcatalog XYZ, but file C is searched for in the current catalog. What this means is that when ENTER is used with the LIBRARY statement, ENTER is a direct substitution for specifying the full pathname in each LIBRARY statement (except the ENTERed catalog must exist at compile-time). So the following two examples are exactly equivalent:

```
library ':abc:def:ghi:jkl:x';  
library ':abc:def:ghi:jkl:y';  
library ':abc:def:ghi:jkl:z';
```

and

```
enter ':abc:def:ghi:jkl';  
library 'x';  
library 'y';  
library 'z';
```

except, in the second case, the catalog :ABC:DEF:GHI:JKL must exist at compile-time if these statements are in a module. Note there may have been side-effects to the ENTER statement in the second example if the LIBRARY statements were INSERT statements.

An Example Module

When writing modules, it is important to make them as self-contained as possible. There should not be a large number of external variables. Any person writing a program or procedure which uses the module should not have access to anything except procedure interfaces to the module. For more information about designing modules, consult The Practical Guide To Structured Systems Design by Meiler Page-Jones (published by Yourdon Press in 1980).

The module below is an example of a good module. There are no variables the user can directly manipulate. All system interfaces are restricted to the module. The user just needs to library this module, declare the procedures, and call them.

```
/* This module implements the hardware/software timer interface
   for the D3 real time clock. */

module timer;          /* define module TIMER */
  dcl time fixed;      /* this LOCAL variable counts D3 ticks */
  time = 0;            /* initialize the time counter */

  when d03int then time = time + 1; /* count D3 ticks */
  enable;              /* enable interrupts */

  get_time: proc returns (fixed) public; /* current time */
    return (time);
  end get_time;

  set_time: proc (new_time) public; /* set starting time */
    dcl new_time fixed; /* new starting time */

    time = new_time;
    write ("3") = 0; write ("3") = 0; /* reset the clock */
  end set_time;

  wait_time: proc (seconds) public; /* wait time SECONDS */
    dcl seconds fixed; /* seconds to wait */
    dcl start_time fixed; /* starting time */
    dcl ticks fixed; /* ticks to wait */

    start_time = get_time; /* get starting time */
    ticks = seconds*200; /* ticks to wait */

    /* wait for TICKS */
    do while (get_time - start_time < ticks); end;
  end wait_time;
end timer; /* end of module TIMER */
```

It would be far easier for the user of a library if the person who writes a module provides an insert file which describes what the library does and how to use it, references the library, and declares all the external variables and procedures. Such an insert file is shown below (the file is called TIMER).

```

/* Timer subroutines:
.
. This library manages the D03 hardware interface, allowing the
. user to deal with timer values (in units of 5 ms) rather than
. the hardware/software interface for the timer.
.
. Three routines are provided:
.   GET_TIME:      returns the current time relative to the
.                  last time set.
.   SET TIME (X):  sets the timer counting from X.
.   WAIT TIME (X): wait X seconds.
.
. WARNING: This library enables interrupts. Do not use this
.           library for time critical applications. */

library ':libs:timer'; /* reference library TIMER */

/* reference these three procedures */

dcl get_time proc returns (fixed) external;
dcl set_time proc (fixed) external;
dcl wait_time proc (fixed) external;

```

It is then trivial for the user to write the following program without any special knowledge of the way the routines are implemented:

```

/* $Dump statistics, $Map memory */

insert ':timer';          /* reference timer routines */

dcl i fixed;

print get_time;           /* print starting time */
call wait_time (5);       /* wait five seconds */
print get_time;           /* verify the ending time */
call set_time (0);        /* initialize the timer */

do i = 0 to 32767; end;   /* increment a variable 32K times */

print get_time;           /* print the elapsed time */
call set_time (0);        /* initialize the timer */

/* count to three, outputting one number every second */

print 'zero ',;
do while (get_time < 200); /* wait until 1st second had passed */
end;
print 'one ',;
do while (get_time < 400); /* wait until 2nd second has passed */
end;
print 'two ',;
do while (get_time < 600); /* wait until 3rd second has passed */
end;
print 'three';
print get_time;           /* print the final time */

```


Section VIII - Hardware Manipulation

This section contains information about using the hardware of the ABLE Series computer. This topic is expanded in the "ABLE Series Hardware Reference Manual".

The Scientific XPL language offers two mechanisms for performing real time input and output. The READ function and WRITE statement operate on 16-bit fixed point quantities and are used to monitor and control interface devices connected to the system.

The READ Function

The READ function is used to transfer a 16-bit data word from an interface device into the user's program. Most such data transfers require only one microsecond to perform. The reserved word READ is followed by a device number enclosed in parentheses.

```
<variable> = read (<device number>);
```

The device number is often expressed in octal (e.g., "34" or "16") although literal declarations can be used to increase program readability. Observe the following two examples:

```
dcl i fixed;
i = read ("34"); /* read data from digital interface device */
print i;

dcl timer literally "'16'"; /* D16 clock */
dcl i      fixed;
i = read (timer) + 2;
print i;
```

The READ function can be used in any arithmetic expression, as if it were a function requiring one fixed point parameter and returning a fixed point result. The device number can also be a variable expression, although in this case the READ will take many microseconds.

The WRITE Statement

The WRITE statement is used to transfer a 16-bit data word from inside the user's program to an interface device connected to the system. The format for the WRITE statement is:

```
write (<device number>) = <expression>;
```

The keyword WRITE is followed by a device number enclosed in parentheses (similar to READ). An equals sign (=) is then followed by any allowed XPL arithmetic expression.

The WRITE statement operates similarly to an assignment statement. The arithmetic expression to the right of the equals sign is first evaluated. The computed number is converted to fixed point automatically if a floating point expression is used. The number is then written out to the specified device.

```
write ("60") = i + 1;  
write ("71") = i;
```

The following example uses a literal to improve program readability:

```
dcl DAC literally "'66'"; /* Digital-to-Analog Convertor */  
dcl i fixed;  
  
do i = 0 to 1023;  
    write (DAC) = i;  
end;  
write (DAC) = 0;
```

Precautions for READ and WRITE

If a program attempts to READ and WRITE data to a device that is not connected to the system (or does not exist), the computer will halt and program execution will not continue. This feature is often used for diagnostic purposes. However, if a Hand Operated Processor is not connected to the system, it will be impossible to tell which device number is being addressed. Without such information, it is very difficult to diagnose errors in the program. The user should be careful to always specify the correct device number in both the READ function and the WRITE statement. Information on device numbers is available in the manual "ABLE Series Hardware Reference Manual".

Assembly Language Programming

It is possible to embed assembly language code within an XPL program using the READ function and the WRITE statement. The source of the instruction is specified in the READ while the destination is specified in the WRITE. The following form would be used for a source to destination instruction:

```
write (<destination>) = read (<source>);
```

For example, to add two fixed point values together:

```
dcl r0    lit '"300"';
dcl r1    lit '"301"';
dcl add0 lit '"210"';

dcl (i, j) fixed;
...
write (r0) = i;           /* I to R0    */
write (r1) = j;           /* J to R1    */
write (add0) = read (r1); /* R1 to ADD0 */
i = read (r0);            /* R0 to I     */
```

The XPL compiler uses the computer's registers in a well-defined way that is described in detail in the "ABLE Series Operating System Reference Manual." In general, registers zero through three are always available for assembly language programming. Furthermore, register thirteen (octal) is never used by the XPL compiler and is available at all times for use by the programmer.

For more information about assembly language programming and a full list of sources and destinations, refer to the "ABLE Series Hardware Reference Manual".

Section IX - Interrupt and Exception Processing

Scientific XPL includes the capability to process many different hardware interrupts with the WHEN statement. Many WHEN statements may appear in one program, and each WHEN statement is used to process a different interrupt or exception. This section contains information about the ABLE Series computer that is explained in detail in the "ABLE Series Hardware Reference Manual".

The WHEN Statement

Interrupts and exceptions are processed in XPL by using the WHEN statement. The format of a WHEN statement is:

```
when <interrupt> then <statement>;
```

The keyword WHEN is followed by an interrupt (or exception) identifier to indicate which interrupt is to be processed by that WHEN statement. The keyword THEN appears after the cell identifier and is followed by any legal XPL statement (usually a BEGIN-END block). WHEN automatically returns from the interrupt at the end of the statement, but can be forced to return with a RETURN statement.

ENABLE and DISABLE

Interrupts are disabled when a program begins execution. In order to use interrupts in a program, interrupts must be explicitly enabled with the ENABLE statement. Interrupts can only be received by the system after the execution of an ENABLE statement and interrupts will be ignored after the execution of a DISABLE statement. Interrupts can be selectively enabled or disabled many times during the course of program execution.

The ENABLE and DISABLE statements take the form:

```
enable; /* interrupts enabled starting here */  
disable; /* interrupts disabled starting here */
```

The two special exception processing WHEN statements WHEN DISKERROR and WHEN BREAK are active at all times and do not require interrupts to be enabled.

Interrupt Processing

The possible XPL interrupt identifiers are shown below.

TTOINT	- terminal output interrupt
TTIINT	- terminal input interrupt
BDB15INT	- custom user device interrupting on BDB15
BDB14INT	- custom user device interrupting on BDB14
DxINT	- interrupt from device X, where X can be any of the following devices (octal): 03, 16, 24, 30, 31, 32, 33, 34, 35, 36, 37 40, 42, 44, 46, 66, 107, 136, 137, 140

Using WHEN statements requires a great deal of care. In complex situations where several devices are simultaneously interrupting the computer, there are endless possibilities for user traps and subtle programming errors. There are some simple rules that should be followed when using WHEN statements that will help minimize such errors.

Interrupts should not be enabled (using ENABLE) inside a WHEN statement. During the processing of a WHEN statement, interrupts are disabled automatically by the system, to avoid the overwhelmingly confusing situation where a device might interrupt its own interrupt processing WHEN statement. Interrupts are reenabled automatically after processing a WHEN statement, as control is returned to the point of interrupt.

Never use a GOTO statement to exit from a WHEN statement. Instead, a RETURN statement should always be used to return control to the point of original interrupt. If another operation must be initiated from within a WHEN statement, the proper means is by flags, queues, and list processing structures.

Swapping procedures should not be called from a WHEN statement.

The following are explanations of some of the available WHEN statements.

WHEN TTOINT

When the terminal interface has finished printing a character on the terminal, a completion interrupt is generated to the computer. If a WHEN TTOINT statement is in the user program and interrupts are enabled, this WHEN statement is activated upon the completion of character transmission. The normal response is to print another character on the terminal using PRINT CHARACTER. If there is no more information to be printed on the terminal then the WHEN statement should simply RETURN. This interrupt is automatically cleared by the XPL runtime system.

WHEN TTIINT

When a character is typed on the computer terminal, a hardware interrupt is generated. This interrupt is passed to the user's WHEN statement if interrupts are enabled when the character is typed.

As the XPL runtime system processes the interrupt, the character is read from the terminal (clearing the interrupt) and stored in a special memory location. The built-in function RCVDCCHARACTER extracts the character from the memory location and returns it to the program (see example below). A typical WHEN TTIINT statement will read the character from RCVDCCHARACTER, examine the character, and branch accordingly.

WHEN D03INT

The D3 Real Time Clock can be used in the interrupt mode. When interrupts are enabled, the D3 generates interrupts at a continuous 200 Hertz rate. This interrupt is automatically cleared by the XPL runtime system.

WHEN D136INT

The D136 Real Time Clock can be used in the interrupt mode. When D136 interrupts are enabled, the D136 generates interrupts at a continuous 200 Hertz rate. This interrupt is automatically cleared by the XPL runtime system.

WHEN D16INT

The D16 Scientific Timer can be used effectively in interrupt driven systems. When the status flag (device "17") of the timer becomes a one, an interrupt is generated if interrupts have been enabled by the ENABLE statement. A typical WHEN D16INT statement would reset the Scientific Timer to interrupt the computer again at an appropriate time interval, and then proceed to increment a timing variable or perform some experimental measurement. This interrupt is automatically cleared by the XPL runtime system.

WHEN D140INT

The WHEN D140INT is used to process interrupts of all types that are generated by the D140 Communications Processor. The D140 is an asynchronous communications subsystem that will support up to 64 terminals. Upon interrupt, the XPL runtime system selects the card and port that generated the interrupt. Separate documentation is available on the D140 entitled "Using the D140 Communications Processor".

WHEN BDB14INT, WHEN BDB15INT

Certain custom devices can be constructed that are to be used in an interrupt mode. These devices will be constructed to ground the Most Significant (BDB15) or Second Most Significant (BDB14) Data Bus Line during the ACKnowledge phase of the interrupt. In this situation, control is automatically transferred to the WHEN BDB14INT or BDB15INT statement, as appropriate.

The following example demonstrates the use of a WHEN statement:

```
when ttiint then begin;
    dcl c fixed; /* character typed by user */

    c = rcvdcharacter;
    print 'You have just typed the character: ', chr (c);
    return;
end;

enable;          /* enable interrupts here before loop */
do while (true); /* infinite loop */
    i = 1;        /* do something here - cannot interrupt an
end;              empty infinite loop */
```

The WHEN statement in the above example processes a terminal input interrupt. Note that the WHEN statement starts with the word WHEN and includes all statements in the BEGIN-END block.

In the example, a hardware interrupt is generated by the serial interface when a character is typed on the computer terminal and interrupts are enabled. If the appropriate WHEN statement is located anywhere in the user program (WHEN TTIINT), the XPL runtime system accepts the hardware interrupt, saves all necessary registers, and transfers control to the statements inside the WHEN statement.

The WHEN statement will read the character that was typed in (using the built-in function RCVDCHARACTER) and then print a message which includes that character on the terminal. After the interrupt has been processed by the WHEN statement, control is returned to the point of original interruption.

Exception Processing

There are two WHEN identifiers that are used to handle special exceptions. These identifiers differ from the others in that interrupts do not have to be enabled for them to work. They are active at all times.

WHEN BREAK

The WHEN BREAK statement is activated if the BREAK sequence is received from the terminal while the PRINT or SEND statement is being used, or while the INPUT or LINPUT statement is being used to read data from the terminal.

WHEN DISKERROR

The WHEN DISKERROR statement is activated if the program tries to read or write a sector of the disk that does not exist, or if a data error is encountered when reading or writing the disk. WHEN DISKERROR is often used to print an appropriate error message on the terminal in the case of a disk error.

The INVOKE Statement

It is sometimes necessary to invoke an interrupt routine in a program when the appropriate interrupt has not occurred. This most often happens when the interrupt routine cannot afford to call a procedure, but the interrupt code needs to be shared. This also occurs when an exception is detected by user software (such as a break typed by the user in some terminal processing code or a bad disk sector found within a user device driver). The INVOKE statement is used to accomplish this:

```
invoke <interrupt>;
```

For example:

```
invoke diskerror; /* a disk error has occurred */
```


SCIENTIFIC XPL REFERENCE MANUAL

Reference Appendices

Appendix A - Built-In Functions

This appendix describes the many built-in functions that Scientific XPL provides. Following a section on devices and storage device input and output, all the built-in functions are listed by category of operation. Following this is an alphabetic listing (sorted by routine name) that describes each routine in detail.

Storage Device I/O

Scientific XPL provides several routines that are used to access storage device media directly. Some of the routines read data from storage devices into internal, external, or polyphonic sampling memory, and others write data from any memory location to storage devices. These routines can be used with any Winchester, floppy, optical disk, or tape that is configured in the system.

The parameters of these routines should be checked carefully, especially with routines that write to devices, because valuable data can easily be overwritten. It is also necessary to make sure that device directories are not destroyed. Refer to the manual "The XPL Catalog Routines" for this directory information before writing data to storage devices.

When using routines to access Winchester systems with more than one drive attached to a device, XPL will automatically determine which physical Winchester disk contains the specified logical sector.

If the storage device you are using with the storage device routines is a SCSI device, you must insert either `:-XPL:SCSI` or `:-XPL:SCSISWAP` into your program to make the device routines operate correctly.

Device Numbers

In order to access storage devices, a numbering convention has been established for all possible storage devices in the system. These numbers should be used to specify which device is to be read from or written to.

- 0: system device (W0 on Winchesters, F0 on floppy systems)
- 1: current device (set under MONITOR by the user with ENTER)
- 2: F0 (Leftmost floppy)
- 3: F1 (Rightmost floppy)
- 4: R0 (Remote floppy)
- 5: R1 (Remote floppy)
- 6: W0 (Winchester disk)
- 7: W1 (Winchester disk)
- 8: T0 (Tape drive)
- 9: T1 (Tape drive)
- 10: O0 (Optical disk)
- 11: O1 (Optical disk)

Only devices that are configured and present in a system should be used. This can be checked with a call to FIND_DEVICE.

Storing Floating Point Data

Special procedures must be followed when accessing floating point data stored on disk. When fixed point numeric quantities are stored, each word of disk storage represents one fixed point number. But floating point quantities are stored inside the computer (and on disk) in an encoded form requiring two words of memory.

The standard routines such as READDATA and WRITEDATA can be used for accessing floating point data stored on disk. Since each data element requires two words, however, the parameter specifying the number of words to transfer should be twice as large as when processing fixed point quantities. Remember that floating point data will require twice as much disk storage also. Note the following examples:

```
declare buf (255) fixed;
call readdata (0, 40, buf, 256);

and

declare buf (255) floating;
call readdata (0, 40, buf, 512);
```

In the first example, 256 words of data corresponding to 256 fixed point numbers are read from sector forty of the system device. In the second example, however, 512 words must be read with READDATA in order to recover only 256 floating point numbers.

XPL Built-In Functions

The XPL built-in functions are listed below by category. Following this list, the routines are explained in detail, listed alphabetically by name.

Storage Device I/O

READDATA - reads data from a device into an array (p. 124)
WRITEDATA - writes data from an array to a device (p. 133)
EXTREAD - reads data from a device into external memory (p. 113)
EXTWRITE - writes data from external memory to a device (p. 115)
POLYREAD - reads data from a device into polyphonic memory (p. 121)
POLYWRITE - writes data from polyphonic memory to a device (p. 122)

Block Manipulation

BLOCKMOVE - copies from one array to another (p. 108)
BLOCKSET - initializes an array (p. 108)
IMPORT - copies from external memory into an array (p. 116)
EXPORT - copies from an array into external memory (p. 112)
EXTSET - initializes a block of external memory (p. 114)

Arithmetic Functions

ABS - absolute value (p. 106)
SQR - square root (p. 129)
SIN - sine (p. 129)
COS - cosine (p. 111)
TAN - tangent (p. 132)
ATN - arctangent (p. 107)
LOG - natural logarithm (p. 119)
EXP - natural anti-logarithm (p. 112)
INT - converts floating point to fixed point (p. 117)

String Functions

BYTE - reads a byte from a string (p. 109)
PBYTE - writes a byte into a string (p. 120)

Bit Manipulation

SHL - shift left (p. 127)
SHR - shift right (p. 128)
ROT - rotate left (p. 125)

Pointers

CORE - an array representing internal memory (p. 110)
ADDR - finds the memory address of a variable (p. 106)
LOCATION - references memory during procedure calls (p. 118)

Program Termination

EXIT - terminates program execution (p. 111)
STOP - halts the computer (p. 130)

Systems Programming

FIND DEVICE - finds a device in the configuration (p. 116)
SET CURDEV - sets the current device (p. 126)
SWAPINIT - reinitializes swapping mechanism (p. 131)
RCVDCHARACTER - gets last character from terminal interrupt
(p. 123)

Hardware

READ - reads a word from an interface device (p. 123)
WRITE - writes a word to an interface device (p. 132)

Alphabetic Listing of Built-In Functions

ABS	computes absolute value	ARITHMETIC
-----	-------------------------	------------

Synopsis: abs: proc (fixed) returns (fixed);
 abs: proc (floating) returns (floating);

Usage: result = abs (num);

where NUM can be either a fixed or floating point number.
 ABS returns the absolute value of NUM.

Example:

```

dcl (i, j) fixed;
dcl (x, y) floating;

i = abs (j);
y = x*y + abs (x);
  
```

ADDR	finds memory address of a variable	POINTERS
------	------------------------------------	----------

Synopsis: addr: proc (scalar variable) returns (pointer);

Usage: ptr = addr (var);

where VAR is any simple variable or subscripted variable.
 ADDR returns a pointer to the absolute location of memory
 in which the variable is stored.

Example:

```

dcl ptr pointer;

ptr = addr (i);           /* simple variable */
ptr = addr (buf (i));   /* subscripted variable */
  
```

See also: LOCATION
 CORE

Synopsis: `atn: proc (floating) returns (floating);`

Usage: `radians = atn (num);`

ATN returns the arctangent of NUM in radians.

XPL does not provide built-in functions for computing arcsine and arccosine, but these functions can be derived from ATN as follows:

$$\begin{aligned}\text{arcsine}(x) &= \text{atn}(x/\text{sqr}(1 - x*x)) \\ \text{arccosine}(x) &= 2*\text{atn}(\text{sqr}((1 - x)/(1 + x)))\end{aligned}$$

Example:

```
decl (x, y) floating;
```

```
x = 2*atn (y);
```

See also: TAN
 SIN
 COS

BLOCKMOVE copies from one array to another BLOCK MANIPULATION

Synopsis: blockmove: proc (fixed array, fixed array, fixed);

Usage: call blockmove (source, dest, length);

This routine copies LENGTH words from the SOURCE buffer to the DESTINATION buffer. The copy works correctly if the two buffers overlap; the destination buffer will always contain the original contents of the source buffer.

The LOCATION and ADDR functions can be used to create pointers to specific array or memory locations.

Example:

```
call blockmove (input buf, string, 8);
call blockmove (buf, loc (addr (data (ptr))), 512);
call blockmove (loc (ptr), buf, len);
```

See also: BLOCKSET

BLOCKSET initializes an array BLOCK MANIPULATION

Synopsis: blockset: proc (fixed array, fixed, fixed);

Usage: call blockset (buffer, length, value);

This routine assigns VALUE to the first LENGTH words of BUFFER.

The LOCATION and ADDR functions can be used to create pointers to specific array or memory locations.

Example:

```
call blockset (input buf, input len, 0);
call blockset (match, 10, false);
call blockset (loc (ptr), 256, 0);
```

See also: BLOCKMOVE

Synopsis: byte: proc (fixed array, fixed) returns (fixed);

Usage: char = byte (string, byte_num);

where STRING is an XPL string and BYTE_NUM is the byte position in that array (starting from zero). BYTE returns the 8-bit value that is stored in that position.

BYTE is usually used with fixed arrays that are XPL strings, as each character is represented by one byte (two characters per word). When using BYTE, the first byte (byte position 0) corresponds to the lower half of word one of the array, because the string length is stored in the first word (element 0) of the array.

BYTE can be used with fixed arrays that are not strings, to manipulate data elements by byte rather than by word. In this case the LOCATION and ADDR functions must be used to access the first word (element 0) of the array, as shown below:

```

dcl list (100) fixed array;

char = byte (loc (addr (list (0)) - 1), byte_num);
    
```

Example:

```

dcl str (10) fixed; /* string of 20 characters */
dcl i          fixed;

linput str; /* get a string from the user */
str (0) = str (0) - 1; /* get rid of carriage return */

do i = 0 to str (0) - 1; /* loop through characters */
    print chr (byte (str, i)); /* print each one */
end;

i = byte (str, 3); /* get the fourth character */
    
```

See also: PBYTE

Synopsis: core: proc (pointer) returns (fixed);
 core (pointer) = expression;

Usage: number = core (ptr);
 core (ptr) = number;

CORE is an array that is used to read and write locations of internal memory. CORE starts at location zero of memory and has as many elements as there are words in memory. It is used in an arithmetic expression with a subscript PTR that represents the word of internal memory to read from or write to.

Care should be taken when writing data to memory that the locations used by the program are not overwritten.

Example:

```
dcl ptr pointer;  
  
x = core (1);            /* get contents of word one */  
core (ptr) = x + y;    /* store a value in location PTR */
```

See also: ADDR
 LOCATION

COS	computes cosine	ARITHMETIC
-----	-----------------	------------

Synopsis: cos: proc (floating) returns (floating);

Usage: result = cos (num);

COS returns the cosine of NUM, where NUM is the angle in radians.

Example:

```
decl (x, y) floating;

x = cos (y);
```

See also: SIN
 TAN
 ATN

EXIT	terminates program execution	PROGRAM TERMINATION
------	------------------------------	---------------------

Synopsis: exit: proc (fixed);

Usage: call exit (status);

where STATUS is the program termination status returned to the Monitor. STATUS should be a zero if the program completed successfully or -1 if the program aborted. Control is returned to the Monitor following the EXIT statement.

Example:

```
when break then call exit (0); /* terminate on BREAK */
...

if find device (dev) = 0 then do; /* DEV not configured */
  print 'Device is not configured in system';
  call exit (-1); /* abort program */
end;
```

See also: STOP

Synopsis: exp: proc (floating) returns (floating);

Usage: result = exp (num);

EXP returns the natural anti-logarithm of NUM. This is equal to e (e = 2.71828...) raised to the power of NUM.

XPL does not provide functions for raising 10 to the power of X or Y to the power of X, but these functions can be derived from EXP as follows:

ten to the x (x) = exp (log (10)*x)
x to the y (x, y) = exp (log (x)*y)

Example:

```
dcl (x, y) floating;  
  
y = exp (x)*10;
```

See also: LOG
 SQR

Synopsis: export: proc (fixed, fixed, fixed array, fixed);

Usage: call export (sector, word, buffer, length);

This routine copies the first LENGTH words from BUFFER into external memory starting at the address specified by SECTOR and WORD (WORD is not restricted to 256).

The LOCATION and ADDR functions can be used to create pointers to specific array or memory locations.

Example:

```
call export (0, 0, buf, 256);  
call export (sector, wd, loc (addr (buf (ptr))), len);
```

See also: EXTSET
 IMPORT

Synopsis: `extread: proc (fixed, fixed, fixed array);`

Usage: `call extread (ms_sector, ls_sector, ext_data);`

where MS SECTOR and LS SECTOR form a 32-bit word pair that identifies the device and sector number that is to be read. The upper eight bits (byte) of this word pair specify the storage device number and the lower 24-bits specify the sector number on that device. EXT DATA is a fixed array which contains the following information:

`ext_data (0) = base sector in external memory`
`ext_data (1) = word offset from base sector`
`ext_data (2) = number of sectors to read`
`ext_data (3) = number of words beyond last sector to read`

Data will be read from a storage device at MS SECTOR and LS SECTOR into external memory. The location in external memory to write to and the amount of data to transfer are both specified in the array EXT DATA. Neither the word offset nor the word length are restricted to 256.

NOTE: If the device you are using with EXTREAD is a SCSI device, you must insert `:-XPL:SCSISWAP` into your program.

Example:

```
dcl info (3) fixed; /* array for using EXTREAD */

info (0) = 104; /* sector 104 of external memory */
info (1) = 0;  /* no word offset */
info (2) = 0;  /* no sectors */
info (3) = file_len; /* read entire file */

/* read file from disk into external memory */

call extread (ms_file_start, ls_file_start, info);
```

See also: EXTWRITE
 READDATA
 WRITEDATA

EXTSET initialize a block of external memory BLOCK MANIPULATION

Synopsis: extset: proc (fixed, fixed, fixed, fixed);

Usage: call extset (sector, word, length, value);

 This routine assigns VALUE to LENGTH words of external
memory starting at the address specified by SECTOR and
WORD. WORD is not restricted to 256.

Example:

 call extset (0, 0, 256, 0);
 call extset (sector, word, len, i);

See also: EXPORT
 IMPORT

Synopsis: find_device: proc (fixed) returns (pointer);

Usage: ptr = find_device (dev_num);

where DEV_NUM is a valid system device number. If a valid device number is passed, and that device is configured in the system, FIND_DEVICE returns an absolute memory pointer to the storage device table entry for that device. Otherwise, it returns a null pointer.

Example:

```
p = find_device (7); /* look for the W1 Winchester */
if p <> null then do; /* the device was found */
    ...
end;
else print 'W1 Winchester disk is not configured.';
```

IMPORT copies from ext memory into an array BLOCK MANIPULATION

Synopsis: import: proc (fixed, fixed, fixed array, fixed);

Usage: call import (sector, word, buffer, length);

This routine copies LENGTH words into BUFFER from external memory starting at the address specified by SECTOR and WORD. WORD is not restricted to 256.

The LOCATION and ADDR functions can be used to create pointers to specific array or memory locations.

Examples: call import (0, 0, buf, 256);
call import (sector, 0, loc (addr (buf (ptr))), len);

See also: EXPORT
EXTSET

INT converts floating point to fixed point

ARITHMETIC

Synopsis: int: proc (floating) returns (fixed);

Usage: result = int (num);

The INT function converts the floating point parameter NUM into a fixed point number, which is returned. NUM is rounded towards negative infinity. That is, positive numbers are truncated (the fractional part is dropped), while negative numbers are set to the nearest more negative integer (int (-1.1) = -2).

Example:

```
dcl (x, y) floating;  
dcl i      fixed;
```

```
i = int (x + y); /* store integer sum in variable I */
```

Synopsis: location: proc (pointer) returns (fixed array);
 location: proc (pointer) returns (floating array);

Usage: call proc_name (location (expression));

The LOCATION function is used to reference absolute locations of memory during a procedure call. LOCATION can only appear in an actual parameter list during a procedure call. The absolute memory location is passed to the procedure as the start of an array (either fixed or floating point). LOCATION can be shortened to LOC.

LOCATION is often used in conjunction with the ADDR function in order to read from or write to an array starting at an element other than element zero.

If a program tries to read a location of memory that does not exist, the computer will halt and the program will stop running.

Example:

```
/* Read a sector from disk into memory starting at
   location 8192. */

call readdata (ms_sec, ls_sec, loc (8192), 256);

/* This example fills a string with nulls starting
   at word one of the string, leaving word zero
   (the string length) intact. */

call blockset (loc (addr (string (1))), str_len, 0);
```

See also: ADDR
 CORE

Synopsis: `log: proc (floating) returns (floating);`

Usage: `result = log (num);`

LOG returns the natural logarithm (i.e., log base e) of NUM. NUM must be a positive non-zero number.

XPL does not provide functions for log base 10 or log base X, but these functions can be derived from LOG as follows:

$$\begin{aligned}\log_{10}(x) &= \log(x)/\log(10) \\ \log_x(x, y) &= \log(y)/\log(x)\end{aligned}$$

Example:

```
decl (x, y) floating;
```

```
x = log (y);
```

See also: EXP

Synopsis: pbyte: proc (fixed array, fixed, fixed);

Usage: call pbyte (string, byte_num, value);

where STRING is an XPL string, BYTE_NUM is the byte position in the array (starting from zero), and VALUE is the number to store there. The lower 8 bits of VALUE will be written to the byte location specified by BYTE_NUM. Only the appropriate upper or lower byte of the array element is altered.

PBYTE is usually used with fixed arrays that are XPL strings, as each character is represented by one byte (two characters per word). When using PBYTE, the first byte (byte position 0) corresponds to the lower half of word one of the array. This leaves the string length in the first word (element 0) of the array.

PBYTE can be used with fixed arrays that are not strings, to manipulate data elements by byte rather than by word. In this case the LOCATION and ADDR functions must be used to access the first word (element 0) of the array, as shown below:

```
    dcl list (100) fixed array;

    call pbyte (loc (addr (list (0)) - 1), byte_num, i);
```

Example:

```
    dcl line (8) fixed array; /* string of 16 characters */
    dcl i      fixed;

    do i = 0 to 15;
        call pbyte (line, i, a.x); /* fill line with X's */
    end;

    call pbyte (line, 9, a.sp); /* put a space in the middle */
```

See also: BYTE

Synopsis: polyread: proc (fixed, fixed, fixed array, fixed);

Usage: call polyread (ms_sector, ls_sector, poly_data, channel);

where MS SECTOR and LS SECTOR form a 32-bit word pair that identifies the device and sector number that is to be read. The upper eight bits (byte) of this word pair specify the storage device number and the lower 24-bits specify the sector number on that device. POLY_DATA is a fixed array which contains the following information:

poly_data (0) = base sector in polyphonic memory
poly_data (1) = word offset from base sector
poly_data (2) = number of sectors to read
poly_data (3) = number of words beyond last sector
 to read

Data will be read from a storage device at MS SECTOR and LS SECTOR into polyphonic sampling memory. The location in polyphonic memory to write to and the amount of data to transfer are both specified in the array POLY_DATA. Neither the word offset nor the word length is restricted to zero. The data will be transferred through the indicated polyphonic CHANNEL (0-31). CHANNEL is usually zero; an interrupt handler could pass the channel being used at the time of interrupt (read ("155")).

NOTE: If the device you are using with POLYREAD is a SCSI device, you must insert :-XPL:SCSISWAP into your program.

Example:

```
dcl info (3) fixed; /* array for using POLYREAD */

info (0) = 10; /* sector 10 of poly memory */
info (1) = 0; /* no word offset */
info (2) = 0; /* no sectors */
info (3) = file_len; /* read the whole file */

/* read file from disk into polyphonic memory */

call polyread (ms_file_start, ls_file_start, info, 0);
```

See also: POLYWRITE
 READDATA
 WRITEDATA

Synopsis: polywrite: proc (fixed, fixed, fixed array, fixed);

Usage: call polywrite (ms_sector, ls_sector, poly_data, channel);

where MS SECTOR and LS SECTOR form a 32-bit word pair that identifies the device and sector number where data is to be written. The upper eight bits (byte) of this word pair specify the storage device number and the lower 24-bits specify the sector number on that device. POLY_DATA is a fixed array which contains the following information:

poly_data (0) = base sector in polyphonic memory
poly_data (1) = word offset from base sector
poly_data (2) = number of sectors to write
poly_data (3) = number of words beyond last sector to write

Data will be written from polyphonic sampling memory to a storage device at MS SECTOR and LS SECTOR. The location in polyphonic memory to read from and the amount of data to transfer are both specified in the array POLY_DATA. Neither the word offset nor the word length is restricted to 256. The data transfer will occur through the indicated polyphonic CHANNEL (0-31). CHANNEL is usually zero; an interrupt handler could pass the channel being used at the time of interrupt (read ("155")).

NOTE: If the device you are using with POLYWRITE is a SCSI device, you must insert :-XPL:SCSISWAP into your program.

Example:

```
dcl info (3) fixed; /* array for using POLYWRITE */  
  
info (0) = ptr; /* sector number in polyphonic memory */  
info (1) = words; /* word offset */  
info (2) = 12; /* read 12 sectors */  
info (3) = 0; /* no extra word length */  
  
/* write data from polyphonic memory to disk location */  
  
call polywrite (f#ms_sector, f#ls_sector, info, 0);
```

See also: POLYREAD
READDATA
WRITEDATA

RCVDCHARACTER gets last character from terminal interrupt SYSTEMS

Synopsis: rcvdcharacter: proc returns (fixed);

Usage: ch = rcvdcharacter;

RCVDCHARACTER returns the last character that was received from a terminal interrupt. It is normally used in a WHEN TTIINT statement (the terminal input interrupt).

Example:

```
when ttiint then begin;
  dcl ch fixed; /* character typed bu user */

  ch = rcvdcharacter; /* get the character */
  print 'You just typed: ', char (ch);
end;
```

READ reads a word from an interface device HARDWARE

Synopsis: read: proc (fixed) returns (fixed);

Usage: i = read (device_number);

The READ function reads a value from the interface module specified by DEVICE NUMBER and returns that value to the program. DEVICE NUMBER can be a constant expression or a variable expression, although the READ will be much slower in the latter case.

If an attempt is made to read a device that is not in the system, the computer will halt.

Example:

```
dcl timer literally '"03"';
dcl i        fixed;

i = read (timer);
```

See also: WRITE

Synopsis: readdata: proc (fixed, fixed, fixed array, fixed);

Usage: call readdata (ms_sector, ls_sector, buffer, length);

where MS_SECTOR and LS_SECTOR form a 32-bit word pair that identifies the device and sector number that is to be read. The upper eight bits (byte) of this word pair specify the storage device number and the lower 24-bits specify the sector number on that device. LENGTH words of data will be read into BUFFER from this device and sector location.

When using READDATA to access Winchester systems with more than one drive attached to a device, XPL will automatically determine which physical Winchester disk contains the specified logical sector.

NOTE: If the device you are using with READDATA is a SCSI device, you must insert either :-XPL:SCSI or :-XPL:SCSISWAP into your program.

Example:

```
dcl buf (256) fixed;
```

```
call readdata (shl (2, 8), 84, buf, 256);
```

In this example, one sector (256 words) of the floppy disk in the F0 drive is read. The number 2 is in the upper byte (device specifier for F0), and the sector address is 84.

See also: WRITEDATA

Synopsis: rot: proc (fixed, fixed) returns (fixed);

Usage: result = rot (number, bit_count);

ROT returns a value that is equal to NUMBER rotated left BIT COUNT bit positions. BIT COUNT must be in the range 0-15. Each bit of VALUE will be shifted to the left BIT COUNT positions, with the most significant bit rotating into the least significant position. Notice the following example, where the number 5000 is rotated 4 positions:

```
      i = 5000;           /* i = 0001001110001000 */
      j = rot (i, 4);     /* j = 0011100010000001 */
```

Example:

```
      dcl (i, j) fixed;

      i = rot (j, 8); /* swap the upper and lower bytes */
```

See also: SHL
 SHR

Synopsis: set_curdev: proc (fixed) returns (boolean);

Usage: if set_curdev (dev_num) then ...

where DEV_NUM is a valid system device number.
SET_CURDEV sets the current device to be the passed device number. If the operation is successful, a TRUE is returned. If an invalid device number is passed, or if the device is not configured in the system, a FALSE is returned and the current device is not changed.

Example:

```
if not set_curdev (dev_num) /* could not change device */
then do;
    print '*** System Error!';
    print 'Could not change current device to ', dev_num;
end;
else do; /* current device was set to DEV_NUM */
    ...
end;
```

Synopsis: shl: proc (fixed, fixed) returns (fixed);

Usage: result = shl (number, bit_count);

SHL returns a value that is equal to NUMBER shifted to the left BIT COUNT bit positions. BIT COUNT must be in the range 0-15. Bits shifted off the left end will be lost, and bits shifted into the right end will be zeros. Notice the following example, where the number 5000 is shifted left 4 positions:

```
i = 5000;          /* i = 0001001110001000 */
j = shl (i, 4);    /* j = 0011100010000000 */
```

Example:

```
dcl dev          fixed; /* device number */
dcl ms_sector    fixed; /* MS word of starting sector */
dcl ls_sector    fixed; /* LS word of starting sector */
dcl sec          fixed; /* number of sectors to read */
dcl buf (2048)   fixed; /* data buffer */

/* The following code sets up a word pair that has a
   device number in the upper byte and the starting
   sector of a file in the lower 24 bits. This is the
   standard way of specifying a file location. */

dev = shl (dev, 8); /* put device in the upper byte */

/* Use AND to make sure that MS_SECTOR has only the
   bottom 8 bits, then OR the device into it. */

ms_sector = (dev or (ms_sector and "377"));

/* Now read the first SEC sectors of the file.
   Notice the word length is given by multiplying the
   sector length by 256, or SHL by 8. */

call readdata (ms_sector, ls_sector, buf, shl (sec, 8));
```

See also: SHR
ROT

Synopsis: shr: proc (fixed, fixed) returns (fixed);

Usage: result = shr (number, bit_count);

SHR returns a value that is equal to NUMBER shifted to the right BIT_COUNT bit positions. BIT_COUNT must be in the range 0-15. Bits shifted off the right end will be lost, and bits shifted into the left end will be zeros. Notice the following example, where the number 5000 is shifted right 4 positions:

```
i = 5000;          /* i = 0001001110001000 */
j = shr (i, 4);    /* j = 0000000100111000 */
```

Example:

```
dcl device fixed; /* device number */
dcl words  fixed; /* number of words */
dcl sectors fixed; /* number of sectors */

/* extract the device from the upper byte of word pair
   identifying file location */

device = shr (f#ms_sector, 8);

/* sectors is words divided by 256, or SHR of 8 */

sectors = shr (words, 8);
```

See also: SHL
ROT

SIN computes sine ARITHMETIC

Synopsis: sin: proc (floating) returns (floating);

Usage: result = sin (num);

SIN returns the sine of NUM, where NUM is the angle in radians.

Example:

```
dcl (x, y) floating;
```

```
x = sin (y)*y;
```

See also: COS
 TAN
 ATN

SQR computes square root ARITHMETIC

Synopsis: sqr: proc (floating) returns (floating);

Usage: result = sqr (num);

SQR returns the square root of NUM, where NUM must be a positive number.

XPL does not provide functions for cube roots or the Xth root of Y, but these functions can be derived from LOG and EXP as follows:

```
cube_root (x)                      = exp (log (x)/3.0)  
xth_root_of_y (x, y) = exp (log (y)/x)
```

Example:

```
dcl (x, y) fixed;
```

```
print sqr (x);  
y = sqr (abs (x));
```

See also: ABS
 LOG
 EXP

Usage: stop;
 stop (value);

The STOP statement is used to halt the computer at a predetermined point in a program. If STOP is used with a fixed point parameter, that VALUE will be written to the Hand Operated Processor (HOP) upon execution of the STOP statement. Pressing the SYNC button on the HOP will cause the program to continue from that point, or control can be returned to the Monitor by pressing the Load button. The STOP statement is most often used for debugging purposes.

Example:

```
dcl status fixed;  
  
status = test_device; /* call testing procedure */  
  
if status <> 0 then do;  
    stop ("10"); /* write a value to HOP to mark this */  
end;
```

See also: EXIT

Synopsis: `swapinit: proc (fixed) returns (fixed);`

Usage: `old_sector = swapinit (new_sector);`

where NEW_SECTOR is the sector address in external memory to set up as the start of the swap file. SWAPINIT reinitializes the swapping mechanism (i.e., resets all the swapping variables), and returns the sector address that was previously the start of the swap file.

SWAPINIT can be used simply to reinitialize the swapping mechanism by using it recursively, as in the second example below. SWAPINIT is called with any value in order to obtain the location of the current swap file, and then SWAPINIT is called again with this returned sector number. The swapping mechanism is initialized, but the swap file is located in the same place it was before.

Note that SWAPINIT only sets a pointer to the swap file, it does not actually put the swap file into external memory. The swap file is normally loaded into external memory whenever a program is loaded into memory by the system.

Example:

```
call swapinit (256); /* swapping starts at sector 256 */  
call swapinit (swapinit (0)); /* reinitialize swapping  
                                mechanism */
```

TAN	computes tangent	ARITHMETIC
-----	------------------	------------

Synopsis: tan: proc (floating) returns (floating);

Usage: result = tan (num);

TAN returns the tangent of NUM, where NUM is the angle in radians. Tangent is equivalent to SIN/COS.

Example:

```
dcl (x, y) floating;
```

```
x = tan (y);
```

See also: ATN
 SIN
 COS

WRITE	writes a word to an interface device	HARDWARE
-------	--------------------------------------	----------

Usage: write (device_number) = value;

The WRITE statement writes a VALUE to the interface device specified by DEVICE_NUMBER. DEVICE_NUMBER can be a constant expression or a variable expression, although the WRITE will be much slower in the latter case.

If an attempt is made to read a device that is not in the system, the computer will halt.

Example:

```
dcl DAC literally "66";
```

```
dcl i   fixed;
```

```
write (DAC) = i;
```

See also: READ

Synopsis: writedata: proc (fixed, fixed, fixed array, fixed);

Usage: call writedata (ms_sector, ls_sector, buffer, length);

where MS SECTOR and LS SECTOR form a 32-bit word pair that identifies the device and sector number that is to be written. The upper eight bits (byte) of this word pair specify the storage device number and the lower 24-bits specify the sector number on that device. LENGTH words of data will be written from BUFFER to this device and sector location.

When using WRITEDATA to access Winchester systems with more than one drive attached to a device, XPL will automatically determine which physical Winchester disk contains the specified logical sector.

NOTE: If the device you are using with WRITEDATA is a SCSI device, you must insert either :-XPL:SCSI or :-XPL:SCSISWAP into your program.

Example:

```
dcl buf (1024) floating;
```

```
call writedata ((shl (7, 8) or 4), 845, buf, 512);
```

In this example, the first 512 words of BUF are written to the W1 Winchester disk (upper byte = 7), starting at sector 262,989 (lower 24 bits = $4 * 65,536 + 845$). WRITEDATA automatically maps this logical sector number to the correct physical Winchester disk on the device.

See also: READDATA

Appendix B - Compilation Control

Compile-time Options

The Scientific XPL compiler provides two compile-time switches that are used to provide information about compilations. If the sequence \$D (or \$d) is found in any comment in the program, statistics about the compilation are "dumped" (hence the D in \$D) to the current output device. This works for modules as well as for the main body of code, and works for both the COMPILE and RUN commands. The current output device is normally the terminal, but can be redirected to the printer by suffixing RUN or COMPILE with ',P' as follows:

run,p	sends output from the RUN to the printer
compile,p	sends output from the COMPILE to the printer

The other compile-time switch is \$M. If the sequence \$M (or \$m) is found in any comment in the program, a symbol table is output after the link process and a sequence number table is output as the compilation completes. The \$M switch produces no output for programs that are RUN or for modules.

The symbol table consists of each library that was linked into the final program in the order it was linked, as well as its starting address in memory, the start of its string and constants area, and the start of its variable or RAM area. Thus, it's a sort of link map (hence the M in \$M). In addition, information about each PUBLIC variable defined in that library is output in alphabetic order by symbol name.

The sequence table consists of a list of line numbers (for each library) with the octal address of the object code created for the statement found at each line number.

It is possible to redirect the symbol table and the sequence table to a file. In order to do this you must save a file named -SYMTAB- in the catalog where you perform your compilations as follows:

```
new -SYMTAB-  
save, <sectors>
```

where <sectors> is the number of sectors you wish to reserve for the symbol table and sequence table output. Ten (10) or twenty (20) is a good number to start with. If -SYMTAB- is not large enough, the output will be redirected to the current output device (the terminal or the printer). If you still want the tables output to a file, you can create a longer -SYMTAB- file and try compiling your program again. The compiler automatically redirects these two tables to the file -SYMTAB- if it exists in the current catalog.

The CONFIGURATION Statement

The CONFIGURATION statement allows the programmer to specify certain system configuration restraints that will be followed during the compilation of a program. The statement takes the form:

```
configuration <symbol>, <symbol>, <symbol>, ...;
```

where any number of <symbol>s selected from the list below can appear:

SMINI	system device is superfloppy diskette
DMINI	system device is double-density diskette
MINI	system device is single-density diskette
MAXI	system device is maxi (8") diskette
MODEL B	CPU Model-B
MODEL C	CPU Model-C
MODEL D	CPU Model-D
MULDIV	use hardware Multiply/Divide unit
NOMULDIV	use software Multiply/Divide routines
MEMORY <size>	absolute memory size (in words)
PTYPE <#>	set terminal type to this number (for PRINT statement)
STYPE <#>	set printer type to this number (for SEND statement)

Terminal and printer type literals are defined in the file SYSLITS in the -XPL programming library (:-XPL:SYSLITS).

For example:

```
configuration dmini, modelB, nomuldiv, memory 40*1024,  
           ptype t#hardcopy;
```

The information provided in the CONFIGURATION statement becomes the default configuration stored in the configuration table of the program. Without a CONFIGURATION statement, the configuration of the program will be the same as the system that it was compiled on (the configuration of that system's Monitor). For more information on the configuration table, see the appendix "Memory Layout".

RAM and PDL Statements

The XPL compiler has the capability to compile programs that will run on a ROM (read only memory) based machine. The RAM statement is used to tell the compiler where the random access memory is located in the computer system memory configuration. The keyword RAM must be followed by a fixed point constant expression, which is the starting address of random access memory:

```
ram <start>;
```

An example of the RAM statement is:

```
ram "1400";
```

If no RAM statement appears, the RAM is assumed to start at the lesser of 8192 or the end of the program's object code.

The XPL compiler uses a Push Down List (PDL) while performing a procedure call. The push down list is used to store information such as automatic variables and return locations during procedure calls and interrupt processing. The amount of memory reserved for the PDL may be changed with the PDL statement. A default length of 256 words is used for the push down list if the program contains no PDL statement. If a program needs more storage for the push down list, as in the case of recursive procedure calls or recursive interrupt processing, its length can be increased using the PDL statement. The keyword PDL is followed by a fixed point expression that sets the word length of the push down list:

```
pdl <size>;
```

An example of to increase the size of the push down list is shown below:

```
pdl 1024; /* set up PDL of 1024 words */
```

In no case can the push down list length be set to zero, or undefined program operation will occur. There is no limit (other than the amount of available memory) on the maximum size of the PDL.

EOF Symbol

The EOF symbol signals the compiler that it is the end of the file. This is used mostly as a debugging aid, for compiling and running a portion of a program. EOF can occur immediately before or immediately after any program statement.

Appendix C - Summary of XPL

Comments and White Space

Comments and white space (spaces and blank lines) can appear anywhere in an XPL program. They are totally ignored by the compiler. Comments start with `/*` and end with `*/`. Comments cannot be nested (i.e., the sequence `/*` cannot appear within a comment).

XPL Data Types

XPL supports two basic data types:

fixed: signed fixed point numbers in the range of -32768 to +32767; can be interpreted as unsigned numbers from 0 to 65535.
floating: floating point numbers in range of 5.5 E-20 to 9.0 E+18.

From fixed point numbers are derived:

boolean: logic value true or false.
pointer: address of another variable or pointer into memory or null.

Constants

XPL supports three fixed point constants: decimal, octal, and hexadecimal. XPL also supports floating point constants. Examples are shown below.

<u>Decimal</u>	<u>Octal</u>	<u>Hexadecimal</u>	<u>Floating</u>
256	"17"	"H0001"	0.001
-142	"00276"	"HFFFF"	-1.04
65530	"377"	"HA2C"	124.675

XPL supports two boolean constants: true and false.

XPL supports one pointer constant: null. All unsigned fixed point numbers are also valid pointer constants.

XPL supports string constants. These are specified by a sequence of up to 128 characters enclosed in apostrophes. An apostrophe within a string constant is specified with two consecutive apostrophes. Examples are 'Hello there.' and 'Don''t touch that dial!'.

Identifiers

Identifiers are used throughout XPL to specify variable names, procedure names, label names, macro names, and module names. An identifier is a sequence of up to 32 letters, digits, and special characters that begins with either a letter or one of the special characters (except period). The special characters are number sign (#), dollar sign (\$), underscore (_), and period (.). There is no distinction made between uppercase and lowercase letters. Examples are `f#ms_sector`, `term_idle`, `answer$`, and `_sys11B_flag`.

Storage Classes

XPL supports four storage classes for variables: automatic, static, public, and external. They are differentiated by scope, lifetime, and when initialized (all XPL variables are initialized to zero). When no storage class is specified, it is assumed to be static (except in recursive procedures where it is assumed to be automatic).

Automatic Variables:

Scope:	Rest of block declared in.
Lifetime:	Life of procedure declared in.
Initialized:	At entrance to procedure declared in.

Static Variables:

Scope:	Rest of block declared in.
Lifetime:	Life of program.
Initialized:	At start of program execution.

Public Variables:

Scope:	Rest of block declared in. Entire program if referenced with EXTERNAL.
Lifetime:	Life of program.
Initialized:	At start of program execution.

External Variables:

Scope:	Rest of block declared in. Used to reference PUBLIC variables.
Lifetime:	No storage associated. Refers to corresponding PUBLIC variable's storage.
Initialized:	No initialization associated since no storage.

Expressions

Expressions are composed of operands (variables, constants, or function calls) combined with operators. The XPL operators are listed below from highest to lowest precedence; operators on the same line are of equal precedence. Operators of equal precedence are evaluated from right to left. Parentheses can be used to force a specific order of evaluation.

```
shr  shl  rot
not
*      /      mod  %      fdiv
+      -
=      ~=     <      <=     >      >=  ieq  ine  ilt  ile  igt  ige
and    or     xor
```

The following symbols are aliases for the operators on the left:

not	~	^
~=	^=	<>
and	&	
or		\

Arithmetic Operators:

+	Addition (binary) or Positive (unary)
-	Subtraction (binary) or Negative/Two's complement (unary)
*	Multiplication
/	Division
mod	Modulus (remainder)
%	Fractional multiply
fdiv	Fractional divide

Signed Relational Operators:

=	Equal to
~=	Not equal to
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to

The value of a relational operation is either true or false.

Unsigned Relational Operators:

ieq	Is equal to
ine	Is not equal to
ilt	Is less than
ile	Is less than or equal to
igt	Is greater than
ige	Is greater than or equal to

The value of a relational operation is either true or false.

Logical Operators:

not	Logical negation
and	Logical and
or	Logical or

The value of a logical operation is true or false. The XPL compiler will evaluate only as much of a logical expression as is necessary to determine if it is true or false. For example, if the first term of an AND expression is false, the rest of the expression will not be evaluated. It is therefore inadvisable to embed a function call that must always occur within a logical expression unless it is the first term; the XPL compiler guarantees to evaluate logical expressions from left to right.

Bit Operators:

not	One's complement
and	Bitwise and
or	Bitwise inclusive or
xor	Bitwise exclusive or
shr	Shift right
shl	Shift left
rot	Rotate left

Statements

XPL programs are constructed of a series of XPL statements. All XPL statements are free format (i.e., many statements can occur on one line or one statement can span multiple lines) and end with a semicolon (;). The simplest statement is the null statement which means do nothing and is specified by just a semicolon.

In the following discussion, "scalar variable" refers to both simple variables and subscripted variables (array elements).

Declaration Statement:

Before an XPL variable can be used, its data type and storage class must be declared.

```
dcl <id> literally <string>;          /* literal */
dcl <id> <type> <class>;                /* scalar */
dcl <id> (<size>) <type> <class>;        /* array */
dcl <id> <type> data <class> (<list>);   /* data */
dcl <id> proc (<parms>) returns (<type>) <class>; /* proc */
dcl <id> label <class>;                /* label */
```

where <id> is an XPL identifier, <string> is a string constant, <type> is an XPL data type, <class> is an XPL storage class (optional), <list> is a comma-separated list of data items or a string constant, and <parms> is a comma-separated list of the data types of the procedure's formal parameters. For procedures, the <class>, if it appears, must be either recursive for a forward reference to a recursive procedure or external to reference a public procedure.

Assignment Statement:

```
<id> = <expression>;
```

where <id> is the identifier of a scalar variable that is assigned the value of <expression>.

Compound Statement:

```
do;          /* compound statement */
  <stmts>;
end;
begin;       /* compound statement; begin new program block */
  <stmts>;
end;
```

where <stmts> is any number of XPL statements. These statements are used to treat a group of statements as one. Note that BEGIN only differs from DO in that BEGIN begins a new level of identifier localization.

Flow of Control:

```
if <expr>                /* IF statement */
then <stmt>;              /* execute if <expr> is TRUE */
else <stmt>;              /* execute if <expr> is FALSE (optional) */
```

where <expr> is a boolean expression and <stmt> is any XPL statement (including the compound statement). The ELSE clause of the IF statement is optional.

```
do while (<expr>);        /* DO while */
    <stmts>;              /* execute while <expr> is TRUE */
end;
```

where <expr> is a boolean expression and <stmts> is any number of XPL statements. The loop is not entered if <expr> is initially FALSE. <expr> is evaluated every time through the loop.

```
do <id> = <expr1> to <expr2> by <expr3>; /* iterative DO */
    <stmts>;                          /* execute while <id> <= <expr2> */
end;
```

where <id> is the identifier of a scalar variable (except an array element), <expr1> is the lower bound of the loop (initial value of <id>), <expr2> is the upper bound of the loop (final value of <id>), and <expr3> is the step (the value to add to <id> each time through the loop). The loop is not entered if <expr2> < <expr1> initially, although <id> will be set to <expr1>. <expr2> - <expr1> must be less than 32768 for the loop to work correctly. Each of the three expressions is evaluated ONCE before entering the loop and never evaluated again. The value of <id> at the end of loop execution is indeterminate. The step is optional, but if present must evaluate to a positive value; if no step is provided, the step will be one. As a special case, a negative constant can be used as a step; the loop executes while <id> >= <expr2>.

```
do case (<expr>);        /* DO case */
    <stmt0>;              /* execute if <expr> = 0 */
    <stmt1>;              /* execute if <expr> = 1 */
    ...
    <stmtN>;              /* execute if <expr> = N */
end;
```

where <expr> is a fixed point expression and <stmt0> through <stmtN> are any XPL statements (including the compound statement). If <expr> igt N, no statements are executed.

```
<id>: <stmt>;            /* program label */
goto <label id>;         /* GOTO statement */
```

where <id> is an XPL identifier, <label id> is an XPL identifier that represents a label and <stmt> is any XPL statement.

Procedures:

```
<id>: proc (<parms>) returns (<type>) <options>; /* proc defn */
    <stmts>;
end <id>;
return (<expr>); /* return from procedure */
```

where <id> is an XPL identifier, <parms> is a comma-separated list of formal parameters, <type> is the type of value returned (e.g., fixed), <expr> is an expression of type <type>, <stmts> is any number of XPL statements, and <options> is any combination of the following (in this order): PUBLIC, RECURSIVE, and/or SWAP. The parameter list, the returns attribute, and the options are all optional. The expression in the RETURN statement must appear in procedures that return values (functions) and must NOT appear in procedures that do not, nor in WHEN statements. Formal parameters are automatic in recursive procedures, static in all others.

```
call <proc id> (<act parms>); /* procedure call */
<id> = <proc id> (<act parms>); /* function call */
```

where <proc id> is an XPL identifier representing a procedure, <act parms> is a comma-separated list of actual parameters. The actual parameter list must match the formal parameters of the called procedure's definition in number and type.

Modules and Libraries:

```
module <id>; /* define a module */
    <stmts>;
end <id>;
```

where <id> is an XPL identifier and <stmts> is any number of XPL statements. In a file that defines a module, no XPL statements can appear outside the module.

```
insert '<treename>'; /* insert source file */
enter '<treename>'; /* change search catalog */
library '<treename>'; /* insert a library */
```

where <treename> is a valid treename. Libraries are executed before the main program and appear in the order in which the compiler finds the LIBRARY statements (nested libraries are linked as they are found).

Interrupts and Exceptions:

```
enable;                /* turn interrupts on */
disable;               /* turn interrupts off */
when <interrupt id> then <stmt>; /* WHEN statement */
invoke <interrupt id>; /* invoke a WHEN statement */
```

where <interrupt id> is one of the several predefined identifiers for interrupt conditions (e.g., TTIINT, D40INT, BREAK) and <stmt> is any XPL statement (including a compound statement). When a program begins execution, interrupts are disabled. The two special exception handlers WHEN BREAK and WHEN DISKERROR are always active and do not require the use of the ENABLE statement. Note that swapping procedures should NOT be called from within a WHEN statement. Interrupts generated by the D50 (TTIINT and TTOINT), the D03, the D136, and the D16 are automatically cleared by the XPL runtime system.

Terminal Input and Output:

```
input <input specifier>; /* get numeric input from user */
linput <string id>;      /* get textual input from user */
```

where <input specifier> is a comma-separated list of identifiers for scalar variables to be input into and <string id> is the identifier for a fixed point array (string) that has 65 elements (0-64; to hold up to 128 characters).

```
print <print specifier>; /* print on terminal screen */
send <print specifier>;  /* send to computer/printer */
```

where the <print specifier> is a comma-separated list of print expressions. A print expression can be any one of the following:

```
<expression>          /* to print a decimal number */
<string constant>     /* to print a string constant */
string (<string id>)  /* to print a string variable */
octal (<expression>)  /* to print an octal number */
chr (<expression>)    /* to print a character by ASCII value */
```

where <expression> is any XPL expression, <string constant> is a string constant, and <string id> is the identifier of an XPL format string (fixed point array). Both PRINT and SEND automatically terminate the output with a carriage return/linefeed pair; this can be disabled by following the last print expression with a comma.

Appendix D - Compiler Error Messages

The efficiency of Scientific XPL derives from the three pass architecture of the Scientific XPL Compiler. Pass 1 of the compiler reads through the source file, performs all the symbol table processing, and creates an intermediate file. Pass 2 creates another intermediate file containing all the instructions for the program. Pass 3 then links in all libraries, throws away dead code, and creates the final object file.

Because the compiler generates very different types of error messages for each pass, the compiler error messages are divided into three sections in this appendix: Pass 1, Pass 2, and Pass 3. The progress of the compiler can be followed by pressing any key on the terminal while a program is compiling. A status message will be printed telling which pass the compiler is currently in and, if appropriate, the line number and file that is currently being processed.

Types of Compiler Errors

Pass 1 will process the main program, as well as any source files that are included with the INSERT statement. Any inserted files are processed as the compiler finds INSERT statements. This pass is where most of the syntax errors will occur. Such things as undefined variables, argument type mismatches, and incorrect statement formats will be found during Pass 1.

Pass 2 has relatively few error messages. If the program is going to be too large for memory, or too many procedures or labels are declared, this pass will flag an error. The line numbers given in these error messages are not completely accurate; an intermediate file is being processed rather than the source file.

Pass 3 is when all libraries are linked into the main program, so these errors tend to concern problems with libraries that are included with LIBRARY statements. Pass 3 detects such things as external variables that do not have a corresponding public definition and libraries that were compiled with a different compiler or for a different processor.

Many messages have a line number and filename appended to them, as in this example:

Incorrect format at line 00023 in file PROG-1

The filename would only be included if the error occurred in an inserted file rather than the main program.

The System Work File .WORK

The compiler uses the system file .WORK, located in the top-level catalog of the system device, to store data as it compiles a program. If .WORK is missing or is not large enough, the compiler will flag an error. This system file should be of the file type DATA and it's length must be a multiple of 8 sectors.

IMPORTANT: If you change the .WORK file at any time, make sure to force a cold boot (type BOOT at the terminal) before doing anything else on your system.

Pass 1 Error Messages

Argument type does not match previous proc defn

The parameter type list of the forward reference declaration for this procedure does not match the parameters of this procedure.

Argument types do not match

The type of an actual parameter in this procedure call does not match the type of the corresponding formal parameter.

Cannot reference global automatic variable

An attempt has been made to access an automatic variable that is declared within the procedure that contains this procedure. Change the global variable's storage class to STATIC or pass it as a parameter to this procedure.

Duplicate definition for '<identifier>'

This identifier has already been declared at the current scope level.

Duplicate 'WHEN' statement

A WHEN statement with the same interrupt identifier (e.g., d03int) as this WHEN statement appears elsewhere in the program. Only one WHEN statement for each interrupt identifier is allowed.

END label does not match

The identifier following the END of a procedure or module definition is not the same as the identifier (name) for this procedure or module. This usually occurs when there are too many or too few END statements within a procedure or module.

Expression not allowed

A variable appears within an expression that must be a constant expression (e.g., the size of an array in a declaration).

Expression too complicated

The compiler has run out of stack space or expression blocks. Use multiple statements and temporary variables to evaluate the expression.

File type mismatch

The file appearing in this INSERT statement is not a text file.

Floating point not allowed in data list

A floating point number appears in a fixed point DATA statement or a floating point number appears in a constant expression that must evaluate to a fixed point value (e.g., the size of an array in a declaration).

Improper recursion

An attempt has been made to call this procedure recursively before all its formal parameters have been declared. Move all formal parameter declarations to the top of the procedure.

Improper type declaration

The RETURNS type of this procedure definition or forward reference is invalid or doesn't match this procedure's forward reference, or a storage class appears incorrectly (e.g., an automatic DATA statement, AUTOMATIC appearing outside a procedure, or formal parameters declared to be public or external, or formal parameters declared to be static in a recursive procedure or automatic in a non-recursive procedure).

Incorrect format

This line contains a syntax error.

Incorrect format in number

A character other than zero through seven appears in an octal constant or a character other than a digit or A through F appears in a hexadecimal constant.

Missing apostrophe

This line is missing the final apostrophe for a string constant.

Missing END statement

A module, procedure, or compound statement is missing an END statement. This is often caused by changing the simple statement of the THEN clause of an IF statement to a compound statement and forgetting to add its corresponding END.

Missing semicolon or format error

A semicolon was not found at the end of the last statement or this line contains a syntax error.

Missing subscript for '<identifier>'

An array reference is missing its subscript. This is often caused by passing an array as an actual parameter to a procedure that expects a scalar value.

'MODULE' must be the first statement of a library

A statement appears before this MODULE statement. Statements cannot appear outside a module.

Module name is missing

This MODULE statement does not include the name of the module.

Multiple 'MODULE' statements not allowed

This file already contains a module statement. Nesting modules or having more than one module in a file is not allowed.

Neither public nor swapping procs can be nested

This public or swapping procedure is defined within another procedure. Public and swapping procedures must be defined at the outermost layer (i.e., not within another procedure).

Nested or non-terminated comment

An attempt has been made to put a comment within a comment or a comment started some lines ago is missing its termination sequence */.

No treename specified

The INSERT, LIBRARY, or ENTER statement at this line does not include a treename.

Not enough arguments supplied

The number of actual parameters in a procedure call is less than the number of formal parameters for that procedure.

Out of symbol table storage

The pass one symbol table is full. Break your program into smaller modules (or get more external memory).

Program too large (interfile1).

The pass one intermediate file is larger than .WORK. Increase the size of the .WORK file.

Program too large (variable space exceeds maximum).

The program uses more than 60K of variable space. Move large arrays to external memory.

Recursive attribute doesn't match forward ref

This procedure is defined to be recursive, but it's forward reference declaration does not include the attribute RECURSIVE, or visa versa.

RETURN statement not allowed

This RETURN statement does not occur within a procedure or a WHEN statement.

Statement outside of module body

This statement appears after the END of a module. Statements cannot appear outside a module.

String constant too long

This string constant is longer than 128 characters.

Subscript not allowed

A subscript appears in the declaration of an external variable, a label, a literal, a data array, a forward reference procedure, or a formal parameter of a procedure.

System error with disk configuration (.WORK is wrong length).

The length of .WORK is not a multiple of eight sectors. Recreate it with a sector length that is a multiple of eight.

System error with disk configuration (.WORK missing).

The file .WORK is not on the system device. Create a .WORK file on the system device.

Too many arguments supplied

The number of actual parameters in a procedure call is greater than the number of formal parameters for that procedure.

Too many END statements

There is an extra END statement. This is often caused by changing the compound statement of the THEN clause of an IF statement to a simple statement and forgetting to remove its corresponding END.

Too many digits in number

More than six octal digits appear in an octal constant or more than four hexadecimal digits appear in a hexadecimal constant or more than eight decimal digits appear to the left or to the right of the decimal point in a floating point constant.

Too many externals declared

More than 4096 external declarations appear in a single source file. Remove unnecessary external declarations.

Too many nested 'BEGIN' statements

BEGIN statements or procedures have been nested beyond the the compiler's capability to deal with them. Change any BEGINS that do not define new localization levels to DOs and unnest procedures.

Too many nested insert files

Insert files have been nested beyond the compiler's capability to deal with them. Redefine the insert file structure so it does not nest as deeply or construct a module from the lower level functions.

Too many numeric constants

The compiler has run out of floating point constant stack space. Use multiple statements and temporary variables to evaluate the expression.

Too many procedures/labels

There are too many procedures (or labels) defined in the source file. Break your program into smaller modules.

Undeclared procedure argument

A formal parameter of this procedure has not been declared.

Undefined symbol '<identifier>'

An attempt has been made to use an identifier before it is declared.

Warning: public variable declared inside a proc

A variable declared within a procedure has been declared to be PUBLIC. This is okay, but the public variable will be undefined if the procedure in which it is declared is never called by the program.

Warning: overwriting contents of data array

One or more elements of a DATA array are being overwritten; DATA arrays are meant to hold constants and should never be altered.

'WHEN' not allowed in procedure

This WHEN statement appears within a procedure. WHEN statements must appear at the outermost layer (i.e., not within a procedure).

Pass 2 Error Messages

Argument types do not match

The type of an actual parameter in this procedure call does not match the type of the corresponding formal parameter.

Expression too complicated at line <line #> (<info>)

The compiler has run out of stack space (<info> is 'push') or expression blocks (<info> is 'get') or automatic temporaries (<info> is 'too many temps in recursive proc'). Use multiple statements and temporary variables to evaluate the expression.

Floating point not allowed with fixed point

An attempt has been made to assign a floating point value to a fixed point variable (this includes decimal constants larger than 65535). Use the INT function if this was intended.

Program too large (pass 2 intermediate file).

The pass two intermediate file is larger than .WORK. Increase the size of the .WORK file.

Program too large (too many variables declared).

This program uses more than 60K of variable space (including temporaries allocated by pass two). Move large arrays to external memory.

Too many numeric constants

The compiler has run out of floating point constant stack space. Use multiple statements and temporary variables to evaluate the expression.

Too many procedures/labels

There are too many procedures (or labels) defined in the source file. Break your program into smaller modules.

Pass 3 Error Messages

Configuration mismatch: Library compiled with Model X processor
[in <library name>] (defined in <library name>).

This library was compiled for a later model processor than the one the main program is being compiled for. Recompile the source module for the same processor model that the main program is being compiled for.

Duplicate definition: <identifier> at line <line #>
[in <library name>] (defined in <library name>).

This identifier defined (i.e., declared to be PUBLIC) in the specified library is redefined at this line in MAIN or in the specified library. Change one of the declarations to EXTERNAL.

Duplicate WHEN statement <when ID>
[in <library name>] (defined in <library name>).

A WHEN statement (specified by the number <when ID>) in the specified library (or MAIN) also appears in another library. Remove one of the WHENs or combine the two.

Library "<library name>" incompatible with current compiler.
Please recompile it.

This library was created by an outdated version of the compiler. Recompile its source module.

Memory conflict - starting RAM address is too low. For this program, the RAM area must be at or above location <#> decimal.

An attempt has been made (with the RAM statement) to set the start of the variable area before the end of the object code. Set the RAM to start at location <#> or above.

Not enough external memory to run program.

The swap file for this program is larger than the amount of external memory in this computer. For this program to run on this computer, change some swapping procedures to non-swapping.

Not enough memory for linker symbol table.

There is not enough internal memory for the linker symbol table. This program cannot be compiled without external memory or more internal memory. Verify the configuration by running the CONFIGUR program.

Not enough memory for Pass3 intermediate file.

This computer does not have enough internal memory to run the XPL compiler. Verify the configuration by running the CONFIGUR program.

Object file too big for compilation.

The internal memory image for this program (or the size of the library being compiled) is larger than 64K; change some non-swapping procedures to swapping (or break this module into smaller pieces).

Program too large for compilation (libraries exceed work file length).

The pass three linker intermediate file is larger than .WORK. Increase the size of the .WORK file.

Program too large for compilation (too many external references).

The external relocation table is full. Redesign your modules to have fewer and tighter interfaces (or get more internal memory).

Program too large for compilation (too many keys).

There is not enough memory to compile this program.

Program too large for compilation (too much swapping scon).

There is not enough internal memory for the swapping string constant relocation table. Change some swapping procedures with many string constants to non-swapping or move some string constants in swapping procedures to DATA arrays.

Program too large (object file/IF collision).

The .WORK file is not large enough to hold both the object file and the intermediate file. Increase the size of .WORK.

Program too large (pass3 object file).

The .WORK file is not large enough to hold both the object file and the intermediate file. Increase the size of .WORK.

Specified library '<library name>' is not a relocatable binary.

The specified library is not the compiled version of a module (file type must be RELOC).

Specified library '<library name>' is zero-length.

The specified library is an empty file.

Specified library '<library name>' is too large.

The specified library is larger than 64K (and thus was not created by the compiler).

System file '.RTB-7' missing.

This file should be in .SYSTEM or top-level system catalog.

System file '.RTC-7' missing.

This file should be in .SYSTEM or top-level system catalog.

Too many public symbols defined.

The pass three symbol table is full. Redesign your modules to have fewer and tighter interfaces (or get more external memory).

Too many public symbols defined (symbol table overflow).

The pass three symbol table is full. Redesign your modules have fewer and tighter interfaces (or get more external memory).

Too many libraries referenced.

The pass three library table is full. Redesign your program to have fewer modules.

Too many libraries referenced (library table overflow).

The pass three library table is full. Redesign your program to have fewer modules.

Types don't match: <identifier> at line <line #>
[in <library name>] (defined in <library name>).

The type of this public identifier (or the parameter type list of this procedure) is different in the specified library (or MAIN) than it was in the defining library. This is usually caused by changing the parameters to a public procedure, but not recompiling all modules that include an external declaration for that procedure (whether they actually use it or not).

Unresolved reference: <identifier> at line <line #>
[in <library name>].

This identifier has been declared EXTERNAL in the specified library, but no corresponding PUBLIC declaration appears in the program.

.WORK not large enough to create swap file.

The combination of the pass three linker intermediate file and the swap file is larger than .WORK. Increase the size of the .WORK file.

Appendix E - Internal Representations

Fixed Point Numbers

Each fixed point data element is a 16-bit, single word quantity. Fixed point numbers can be interpreted as signed integers in the range of -32,768 to +32,767 or as unsigned integers in the range of 0 to 65,535. Both signed and unsigned integers are processed and stored identically inside the computer; the difference lies in the user interpretation of the bit patterns that are stored in memory.

The bits that make up a fixed point number are labeled 0 through 15 as shown in the following diagram.

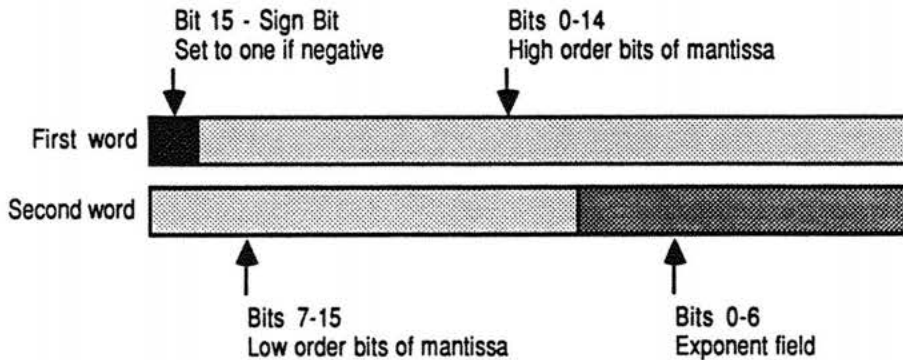


Negative integers are stored in memory in two's complement form. Bit 15 is the sign bit and will be set to a one in the case of a negative quantity. To obtain the negative value of a number in two's complement form, take the one's complement of the number (invert each bit) and then add one. Two's complement numbers are defined in such a way that when a positive number is added to its negative counterpart the result is a 16-bit word of all zeros. Note the following examples:

0000000000000011	=	3
0000000000000010	=	2
0000000000000001	=	1
0000000000000000	=	0
1111111111111111	=	-1
1111111111111110	=	-2
1111111111111101	=	-3

Floating Point Numbers

Floating point numbers are signed reals in the range of plus and minus 5.5 E-20 to 9.0 E18. They are stored in two consecutive locations of memory in a compacted form as shown in the following diagram:



Floating point variables are stored in terms of a sign bit, a 24-bit mantissa, and a 7-bit exponent field. The sign bit is a zero in the case of a positive number, or a one in the case of a negative number. To prevent multiple representations of the same number, the mantissa is always normalized so that its most significant bit is a one, except in the case of the number zero where the sign, mantissa, and exponent field are all zeros. The binary point is always located to the left of the most significant bit of the mantissa.

The exponent field represents a power of two exponent that is used to scale the mantissa up to 64 places left or right. The exponent field is stored in excess-64 notation so that an exponent field of all zeros represents -64 and all ones represents +63.

For example, the floating point number -25.0 is represented internally as follows:

```
-25.0000 = 111001000000000000000000
           0000000001000101
```

and is interpreted (in binary) like this:

```
Sign bit = 1
Mantissa = .110010000000000000000000
Exponent = 1000101
```

Computing the decimal equivalent of this number is done in the following way:

```
Mantissa = (1*0.5) + (1*0.25) + (1*0.03125) = 0.78125
Exponent = 69 - 64 = 5
Mantissa with exponent = 0.78125 * 32 = 25.00
Result with sign bit = -25.00
```

The following table presents the internal bit format for some floating point numbers.

<u>Number</u>	<u>Internal Format</u>
0.000000	0000000000000000 0000000000000000
1.000000	0100000000000000 0000000001000001
0.500000	0100000000000000 0000000001000000
25.00000	0110010000000000 0000000001000101
0.100000	0110011001100110 0110011000111101

Character Strings

Character strings can be stored in fixed point arrays, using a special format that makes manipulating strings convenient for the programmer. An array that contains ASCII characters has the string length stored in the first word (element zero), and the string characters stored in the rest of the array, starting with element one. The zeroeth element of the array contains the number of used 8-bit bytes in the array, with each byte of the array representing one ASCII character. Each 16-bit array element therefore contains two bytes (two characters), the lower half being an even byte, and the upper half being an odd byte:

Array (0)	Character length of string	
Array (1)	Byte 1	Byte 0
Array (2)	Byte 3	Byte 2
Array (3)	Byte 5	Byte 4
etc.		

Character strings in this standard format can be easily read from and written to the terminal using the LINPUT and PRINT statements. There are also two built-in functions (BYTE and PBYTE) that are used to process textual string information.

Appendix F - Memory Layout

This appendix describes the layout of internal and external memory while an XPL program is executing, and how that program can access and use different areas of memory. There are three basic categories of information in this appendix: the disk image of a program, the internal memory structure, and the external memory structure.

Every ABLE computer has some amount of internal memory, from 16K up to 64K words. The internal memory size of your computer must be specified for the Monitor with the CONFIGUR program. External memory can be purchased as an option and does not need to be specified in the system configuration. The computer simply uses external memory if it finds it in the system. Note that a program which requires external memory (i.e., one that has swapping procedures) will not run on a system that does not have external memory.

If you are going to use the information in this section for programming purposes, you must have a copy of the -XPL programming library on your system. The -XPL file SYSLITS (: -XPL:SYSLITS) contains the system literals that will allow you to access sections of memory during program execution. This programming library can be obtained from New England Digital Customer Service.

It is important to use the system literals in SYSLITS for all programs that access memory. The actual locations used to store things in memory often change with new software releases and SYSLITS is updated to accommodate these changes. Always use the version of -XPL that is compatible with the software version you are running to make sure you are accessing the correct memory locations.

SYSLITS provides the basic information needed to use the system literals. For additional information, refer to the manual "ABLE Series Operating System Reference Manual".

The Configuration Table

The configuration table is a special area of memory that contains information about the system. There is a copy of the configuration table resident in internal memory at all times. There is also a configuration table stored on disk with every program. The table contains information such as how much memory is in the system and what storage devices are attached to the system. When a program is run, the system's configuration is copied into the program's configuration table.

The format of the configuration table is outlined in the -XPL file SYSLITS. The format of the table is the same on disk as it is in memory. There is a pointer to the start of the configuration table which can be obtained by using the literal C#CONTAB. Using this pointer, all the information stored in the table is available by using other literals. For example, the literal C#STKLEN can be used to find out how much internal memory is used for the program's runtime stack.

The Swap File

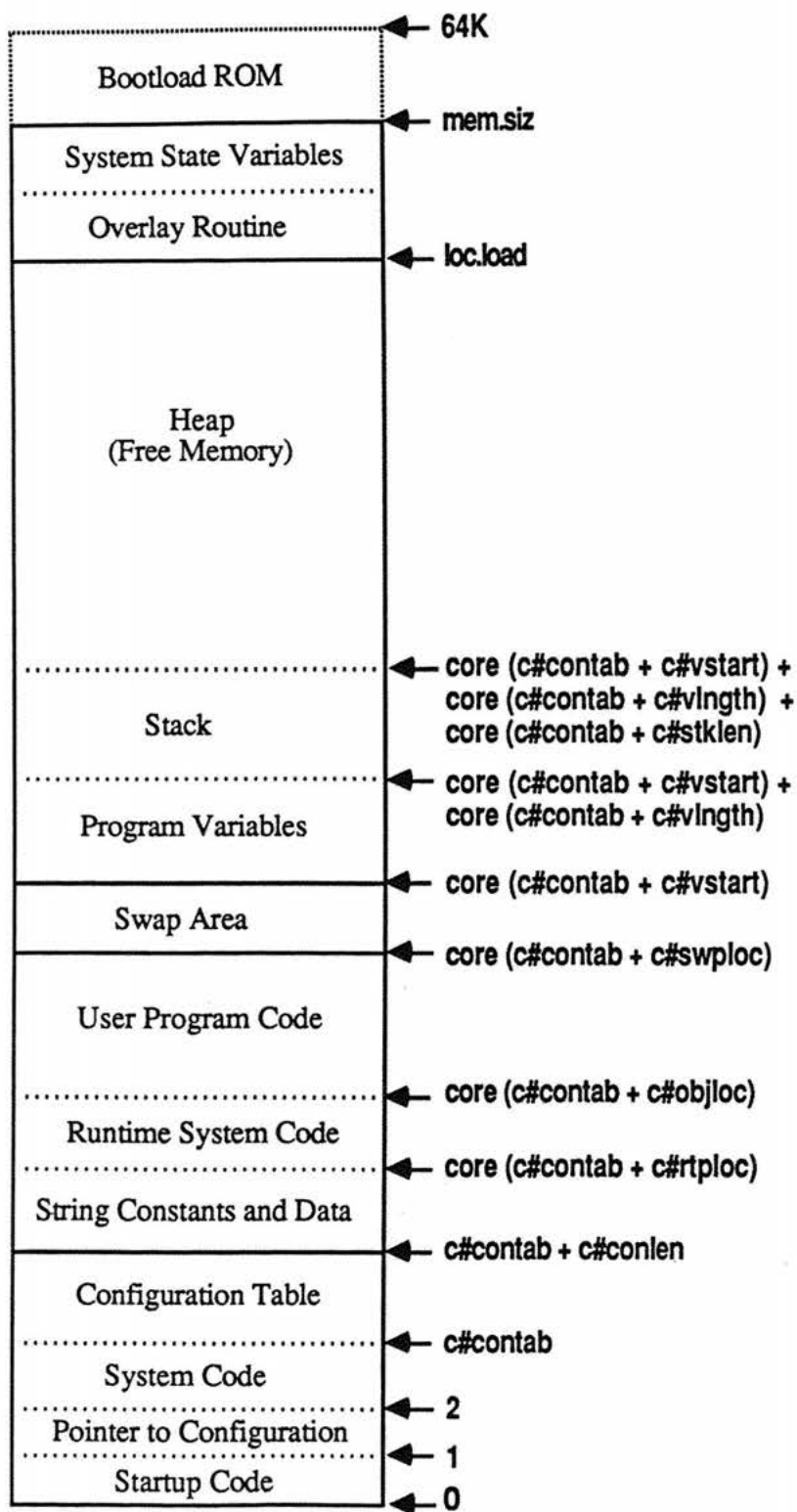
All procedures that are declared to be swapping are stored in external memory until they are needed by the program. When a program is compiled, the last section of the compiled code contains the swap file for the program. The swap file contains the code for all the swapping procedures, as well as a lookup table with the location of each swapping procedure (the swap lookup table).

When a program is run, the swap file is copied into external memory. A section of internal memory called the swap area is reserved to hold any procedure that is swapped into internal memory. The swap area is as large as the largest swapping procedure. Only one swapping procedure is resident in internal memory at any one time.

Memory Image: Internal

On the following page is a detailed diagram of internal memory while a program is executing. Located out to the right of each section are the system literals you would use to find each area during program execution. The pointer to the configuration table is found by using the literal C#CONTAB.

The word size of useable internal memory in your system is set up automatically in the variable MEM.SIZ when you insert :-XPL:SYSLITS into a program. For example, if you have the full 60K in your system, MEM.SIZ will be 61440 words (60*1024).



The upper part of useable memory is reserved for the system state variables and the overlay routine. The system state variables contain information about the current file, the current system and user catalogs, and other information that needs to be preserved when overlaying from one program to another. The literals for accessing these system variables are in SYSLITS.

The internal memory not explicitly used by the program (the heap) can be used by the user program. Access to this memory is faster than access to external memory, so the heap is often used as a transfer buffer for copying files from one place to another or as a data storage area when time consuming operations are being performed. The following program segment figures out how much free memory is available and uses that area as a buffer.

```

insert ':-xpl:syslits'; /* get the system literals */

dcl free_start fixed; /* start of free memory */
dcl free_end fixed; /* end of free memory */
...

/* The start of the heap is found by starting at the program
   variable area, adding the length of the variable area to
   that, then adding the length of the stack. */

free_start = core (c#contab + c#vstart) +
             core (c#contab + c#vlength) +
             core (c#contab + c#stklen);

free_end = loc.load; /* end of free memory */

total_free = free_end - free_start; /* word length of heap */
sectors = shr (total_free, 8); /* sector length of heap */

/* Recompute the word length of heap so that it is a
   multiple of a sector boundary (for using it with READDATA
   and WRITEDATA) */

total_free = shl (sectors, 8);

/* The following call to READDATA uses the LOCATION function
   to read data from disk into free memory. */

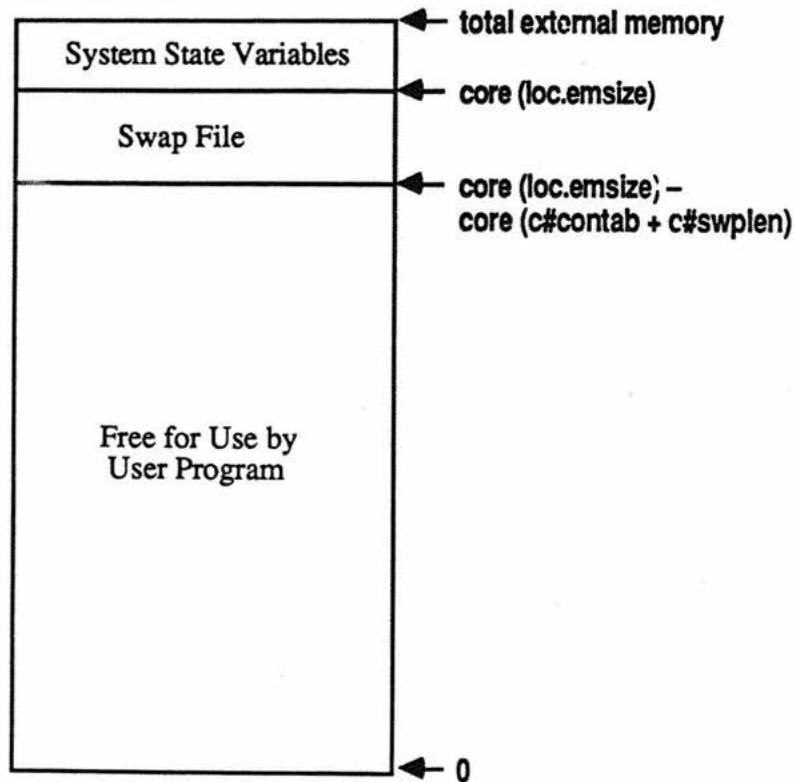
call readdata (ms_sec, ls_sec, loc (free_start), total_free);

```

Memory Image: External

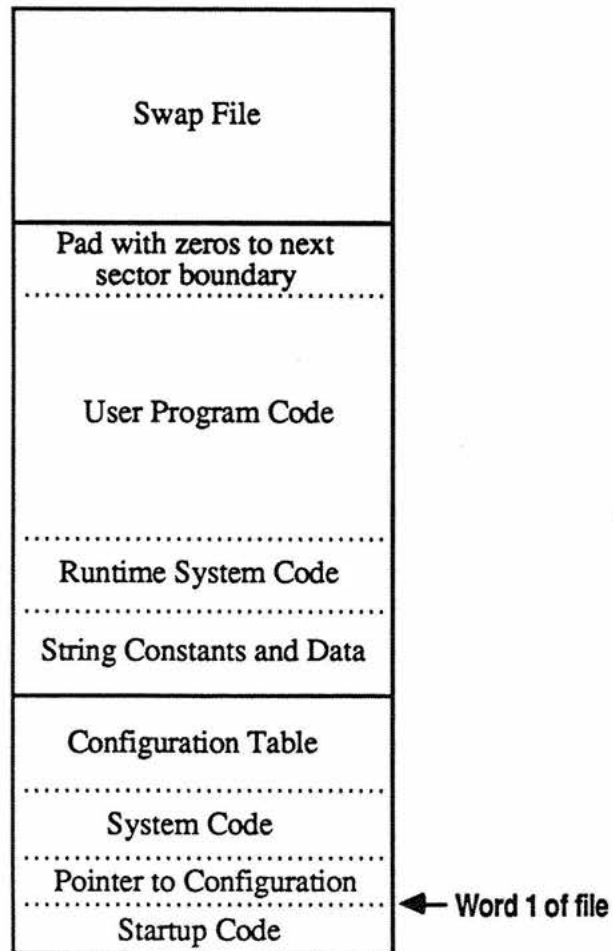
The diagram below shows the layout of external memory while a program is executing. As with internal memory, a small area at the end is reserved for system state variables. Unlike internal memory, where locations and lengths are specified in words, external memory is divided into sectors. The literal LOC.EMSIZE is used to find the number of useable sectors of external memory.

All of external memory except for the state variable area and the swap file is free for user applications, starting at sector zero. There are several built-in routines that are provided for reading from and writing to external memory. See the appendix "Built-in Functions" for descriptions of how to use these routines.



Disk Image

All compiled programs on disk are stored in a particular format. This format is similar to the structure of internal memory while a program is running. The following diagram illustrates this format:



The configuration table for a program can be examined by looking at the first sector of the program. Currently the configuration pointer for a program is located in the word at address one (check SYSLITS to make certain this is the case for your version of software). By using this pointer and the rest of the configuration literals in SYSLITS, information about the memory requirements of the program can be obtained. The literal C#OBJLEN can be used to get the length of the code up to the end of the user program code.

The following program segment reads the first sector of a program on disk, then uses SYSLITS to find out the length of the swap file for that program.

```
insert ':-xpl:syslits';

dcl buf (255)    fixed; /* buffer for first sector of file */
dcl contab      pointer; /* pointer to configuration table */
dcl swap_sectors fixed; /* number of sectors for swap file */
...

/* read the first sector of the program saved on disk */
call readdata (f#ms_sector, f#ls_sector, buf, 256);

contab = buf (1); /* get pointer to the config table */

/* The sector length of the swap file is located in a word
   offset defined by C#SWPLEN. So by adding C#SWPLEN to the
   start of the configuration table, we find the length of
   the swap table. */

swap_sectors = buf (contab + c#swplen);
```


Appendix G - D4567 Operation

The Hardware Multiply/Divide Unit (D4567) can be used with the READ function and the WRITE statement to perform multiplications and divisions of unsigned integers at a higher rate than is possible with the multiply (*) and divide (/) operators. This is possible because there is no sign correction that is required when using unsigned integers, while the sign correction routine is automatically invoked with * and /. Of course, this feature is only available when the optional Hardware Multiply/Divide Unit is connected to the computer.

The computer accesses the Multiply/Divide Unit by using READ function and the WRITE statement specifying devices 4, 5, 6, and 7. For programming purposes, the Hardware Multiply/Divide Unit can be viewed as two sixteen bit registers that can be loaded, read, and operated on by the computer. These two registers are called the A register and the B register and each can be read or written by the computer specifying device addresses 4 and 5, respectively. The A register (device 4) is cleared whenever the B register (device 5) is loaded in order to speed up both the multiplication and division operations. The advantage of this feature will become evident shortly.

Device addresses 6 and 7 are used to issue commands to the unit. Writing a fixed point number to device 6 directs the multiplier section to multiply the unsigned 16-bit number in the B register (device 5) by the unsigned 16-bit number that is being written to device 6. An unsigned 32-bit product is first formed and then added to the contents of the A register (device 4). The upper 16 bits of the result can then be accessed by reading the A register (device 4), while the lower 16 bits of the result are available in the B register (device 5).

If the programmer is only using the lower 16 bits of the result (device 5), then the computed number is correct for both signed and unsigned integers as long as the two operands were within range. No sign correction is required in this case.

A division is performed by first loading a 32-bit divisor into the A register (MS word) and B register (LS word), then writing a fixed point number to device 7. An integer unsigned division is performed in 1.8 microseconds producing a quotient in the B register (device 5) and a remainder in the A register (device 4). The following program segments demonstrate the use of the Multiply/Divide Unit.

```

declare (i, j) fixed;

write (5) = i;    /* load B register and clear A */
write (4) = 2;    /* load A register with addend */
write (6) = j;    /* multiply I times J and add 2 */
i = read (4);     /* upper 16 bits in A register */
j = read (5);     /* lower 16 bits in B register */

write (5) = j;    /* for divide, always load lower bits first */
write (4) = i;    /* load upper 16 bits of divisor */
write (7) = 100; /* write dividend to device 7 */
i = read (5);     /* quotient in the B register */
j = read (4);     /* remainder in the A register */

```

Warning: There is a limitation in the D4567 that requires the programmer to read the result out of register B (device 5) between commands, whether the result is needed or not. The following sequence is illegal and will produce erroneous results:

```

/* evaluate 1*7/12 in full 32-bit resolution */

write (5) = i;
write (6) = 7;
write (7) = 12;
i = read (5); /* pick up result */

```

The correct sequence is:

```

write (5) = i;
write (6) = 7;
i = read (5); /* this must be here !! */
write (7) = 12;
i = read (5); /* pick up result */

```


INDEX

- \$D compile-time switch, 135
- \$M compile-time switch, 135
- % (fractional multiply), 25, 26
- * (multiplication), 25, 26
- + (addition), 25
- + (unary plus), 25
- (subtraction), 25
- (unary minus), 25
- SYMTAB- file, 135
- XPL programming library, 101, 136, 163
- .WORK file, 148
- / (division), 25, 26
- 32-bit precision, 26, 27, 171
- < (less than), 29
- <= (less than or equal), 29
- <> (not equal), 29
- = (equal), 29
- > (greater than), 29
- >= (greater than or equal), 29
- ^= (not equal), 29
- ~= (not equal), 29

- ABS function, 106
- Absolute value (see ABS)
- Actual parameters, 56
- Addition, 25
- ADDR function, 50, 106
 - with LOCATION, 51
- Aliases for operators, 141
- AND, 31
 - bit operator, 32
 - logical operator, 31
 - parentheses with, 36
 - symbols for, 32
 - truth table of, 34
- Anti-logarithm, 112
- Appendices, 99
- Arccosine, 107
- Arcsine, 107
- Arctangent (see ATN)
- Arguments (see Parameters)
- Arithmetic functions, 104
 - ABS, 106
 - ATN, 107
 - COS, 111
 - EXP, 112
 - INT, 117
 - LOG, 119
 - SIN, 129
 - SQR, 129
 - TAN, 132
- Arithmetic operators, 25
 - summary of, 141
- Array index (see Subscripts)
- Arrays, 45
 - CORE array, 50
 - data list, 49
 - declaration of, 45
 - declared as parameters, 57
 - in procedures, 57
 - LOCATION, 51
 - out of bounds, 46
 - passed by reference, 59
 - passed to procedures, 51
 - reading from, 46, 50
 - storing data in, 46, 50
 - strings, 47
 - subscripts, 45
- Assembly language, 91
- Assignment statement, 35, 143
- ATN function, 107
- ASCII characters (see Strings)
- AUTOMATIC storage class, 69
 - access of, 69
 - as default, 70
 - push down stack, 69
 - summary of, 140

- BEGIN statement, 37
- Binary arithmetic operators, 25
- Bit manipulation, 32, 104
 - AND, 32
 - NOT, 32
 - operators, 32
 - OR, 32
 - ROT, 33, 125
 - SHL, 33, 127
 - SHR, 33, 128
 - summary of, 142
 - XOR, 32
- Bit operators, 32
 - summary of, 142
- Black box, 78
- Block manipulation, 104
 - BLOCKMOVE, 108
 - BLOCKSET, 108
 - EXPORT, 112
 - EXTSET, 114
 - IMPORT, 116
- Block structure, 64

- BLOCKMOVE function, 108
- BLOCKSET function, 108
- Boolean data type, 14
 - FALSE, 14
 - logical operators, 31
 - relational operators, 29
 - TRUE, 14
 - use of, 63
- BREAK (see WHEN BREAK)
- Built-in functions, 101
 - alphabetic list, 106
 - categorical list, 104
- BYTE function, 48, 109
 - using literals for, 18, 89
- Control-Q, 20
- Control-S, 20
- Conversions, 35
 - numeric, 35
 - of parameter types, 60
- CORE array, 50, 110
- COS function, 111
- Cosine (see COS)
- Current catalog, 76
- Current device, 126
- Custom interrupts, 96
- D3 Real Time Clock, 95
- D16 Scientific Timer, 95
- D136 Real Time Clock, 95
- D140 Communications Processor, 96
- D4567 (see Hardware Multiply/Divide)
- Dangling ELSE clause, 39
- DATA declaration, 49
- Data types, 13
 - boolean, 14
 - conversion of, 35
 - fixed point, 13
 - floating point, 13
 - pointer, 14
 - summary of, 139
- DCL, 17
- Decimal point, 15
- DECLARE statement, 17
- Declarations, 17
 - AUTOMATIC, 69
 - arrays, 45
 - arrays as parameters, 57
 - data list, 49
 - DECLARE statement, 17
 - EXTERNAL, 81
 - formal parameters, 56
 - global, 53
 - labels, 67
 - literals, 18
 - PUBLIC, 80
 - procedures, 72
 - STATIC, 69
 - summary of, 143
 - variables, 17
- Device directories, 101
- Devices, 94
 - device numbers, 89
 - device specifiers, 74
 - storage devices, 101
- Directories, 101
- CALL statement, 55, 56
- CASE statement (see DO CASE)
- CHARACTER, 20
- Character output, 20
- Character strings (see Strings)
- CHR, 20
- Combining expressions, 31
- Comments, 9, 53, 54, 139
- Compilation control, 135
 - compile-time options, 135
 - CONFIGURATION, 136
 - EOF symbol, 137
 - PDL, 137
 - RAM, 137
- Compile-time options, 135
- Compiler, 147
 - XPL compatibility, 163
 - .WORK file, 148
 - efficiency of, 147
 - Pass 1 errors, 148
 - Pass 2 errors, 154
 - Pass 3 errors, 155
 - structure of, 147
 - types of errors, 147
- Compound statement, 37, 143
- Conditional execution (see IF)
- CONFIGUR program, 163
- Configuration of Monitor, 136
- CONFIGURATION statement, 136
- Configuration table, 164, 169
- Constants, 14
 - data list, 49
 - fixed point, 14
 - floating point, 15
 - hexadecimal, 14
 - octal, 14
 - precomputation of, 25, 28
 - string, 15
 - summary of, 139

DISABLE statement, 93
 Disk image of a program, 168
 Disk storage (see Storage device I/O)
 DISKERROR (see WHEN DISKERROR)
 Division, 25, 26
 DO CASE statement, 42
 DO loop, 40
 DO statement, 37
 DO WHILE loop, 39
 Dump option, 135
 Dynamic variables (see AUTOMATIC)

ELSE clauses, 38
 ENABLE statement, 93
 End of file (see EOF)
 END statement, 37, 55, 79
 ENTER statement, 76
 asterisk with, 76
 with INSERT, 77
 with LIBRARY, 84
 EOF symbol, 137
 Example module, 86
 Example program, 10
 Exceptions, 93, 97
 summary of, 146
 Exclusive or (see XOR)
 EXIT function, 111
 EXP function, 112
 EXPORT function, 112
 Expression evaluation, 31
 Expressions, summary of, 141
 External memory, 163
 built-in functions for, 104
 description of, 163
 memory image, 167
 state variables, 167
 swap file, 164
 swapping procedures, 71
 EXTERNAL storage class, 81
 summary of, 140
 EXTREAD function, 113
 EXTSET function, 114
 EXTWRITE function, 115

FALSE, 14
 FDIV (fractional divide), 25, 26
 FIND DEVICE function, 116
 Fixed point data type, 13
 constants, 14

 conversions, 35
 internal format, 159
 overflow, 25, 26, 29
 range of, 13
 Floating point data type, 13
 constants, 15
 conversions, 35
 internal format, 160
 overflow, 25, 27
 range of, 13
 storing data, 103
 Floppy disk (see Storage device I/O)
 Flow of control, 37
 summary of, 144
 Formal parameters, 56
 Forward reference procedures, 72
 Fractional divide, 25, 26
 Fractional multiply, 25, 26
 Free memory (see Heap)
 Functions, 62
 built-in, 101
 used in expressions, 63

Global declarations, 53
 GOTO statement, 44, 61, 68

Hand Operated Processor (see HOP)
 Hardware manipulation, 89
 assembly language, 91
 Hardware Multiply/Divide, 171
 interface devices, 89
 interrupt processing, 93
 READ, 89, 123
 WRITE, 90, 132
 Hardware Multiply/Divide, 171
 limitation of, 172
 speed of, 172
 with arithmetic operators, 27
 Heap (free memory), 166
 Hexadecimal constants, 14
 HOP, 90, 130

Identifiers, 16
 summary of, 140
 IEQ, 30
 IF statement, 38
 IGE, 30
 IGT, 30
 ILE, 30
 ILT, 30

- IMPORT function, 116
- INE, 30
- Infinite loop, 18, 40
- Initialization of variables, 69, 80
- INPUT statement, 21
- INSERT statement, 75
 - with ENTER, 77
- INT function, 35, 61, 117
- Interface devices, 89
- Internal formats, 159
 - fixed point, 159
 - floating point, 160
 - strings, 161
- Internal memory, 50
 - ADDR, 50
 - conserving, 71
 - CORE array, 50
 - description of, 163
 - getting a pointer to, 50
 - LOCATION, 51
 - memory image, 164
 - overlay routine, 166
 - reading from, 50
 - state variables, 166
 - swap area, 164
 - swapping procedures, 71
 - writing to, 50
- Internal representations, 159
- Interrupts, 93
 - default state of, 93
 - DISABLE, 93
 - ENABLE, 93
 - identifiers for, 94
 - in WHEN, 94
 - INVOKE, 97
 - summary of, 146
 - swapping procedures, 94
 - WHEN statement, 93
 - with PRINT, 20
- Introduction, 7
- INVOKE statement, 97
- Iterative DO loop, 40
 - linking of, 82
 - order of, 82
 - referenced from modules, 84
 - summary of, 145
- LIBRARY statement, 82
 - with ENTER, 84
- Line numbered files, 75
- LINPUT statement, 22
- Link map, 135
- Linking, 79, 82
- LIT, 18
- LITERALLY, 18
- Literals, 18
- LOC, 51
- Local declarations, 54
- LOCATION function, 51, 118
 - with ADDR, 51
- LOG function, 119
- Logarithm (see LOG)
- Logical operators, 31
 - summary of, 142
- Loop variables, 40, 46
- Loops, 37
 - DO WHILE, 39
 - infinite, 40
 - iterative DO, 40
 - loop variables, 40, 46
 - negative increments, 41
- Macros (see Literals)
- Memory, 50, 163
 - built-in functions for, 104
 - conserving, 71
 - external memory, 101, 163
 - internal memory, 101, 163
 - layout of, 163
 - pointers into, 50
 - polyphonic, 101
 - swapping procedures, 71
- MOD function (remainder), 26
- Modular programming (see Procedures)
- MODULE statement, 79
- Modules, 73, 78
 - compilation of, 79
 - END statement in, 79
 - ENTER statement, 76
 - example of, 86
 - EXTERNAL storage class, 81
 - INSERT statement, 75
 - linking of, 79
 - PUBLIC storage class, 80
 - scope, 80, 81
- Keywords, 16
- Labels, 43, 44
 - declaration of, 67
- Libraries, 73, 82
 - existence at compile-time, 84
 - LIBRARY statement, 82
 - link map for, 135

- summary of, 145
- with WHEN, 79

Modulus function (see MOD)

Multiplication, 25, 26

Natural anti-logarithm, 112

Natural logarithm, 119

Negation, 25

Negative loop increments, 41

Nesting, 55

- INSERT statements, 75
- comments, 53
- libraries, 82
- modules, 79
- procedure calls, 63
- program blocks, 68
- with AUTOMATIC, 69

NOT, 31

- bit operator, 32
- logical operator, 31

NULL, 14

Null statement, 42

Numeric conversions, 35, 60

Numeric input, 21

OCTAL, 21

Octal constants, 14

Octal output, 21

One's complement, 32, 159

Operating system, 163

Operators, 25

- aliases for, 141
- arithmetic, 25
- bit, 32
- logical, 31
- relational, 29
- summary of, 141
- unsigned relational, 30

Optical disk (see Storage device I/O)

OR, 31

- bit operator, 32
- logical operator, 31
- parentheses with, 36
- symbols for, 32
- truth table of, 34

Order of evaluation, 36

Overlay routine, 166

Parameters, 55, 56

- actual, 56

- formal, 56
- pass by reference, 59
- pass by value, 58

Parentheses, 36

Parity with LINPUT, 23

Pass 1 compiler errors, 148

Pass 2 compiler errors, 154

Pass 3 compiler errors, 155

Pass by reference, 59

Pass by value, 58

Pathnames, 74

PBYTE function, 48, 120

PDL statement, 69, 137

PL/I, 7

Pointer data type, 14, 50

Pointer functions, 105

- ADDR, 106
- CORE, 110
- LOCATION, 118

Polyphonic memory, 101

- built-in functions for, 104

POLYREAD function, 121

POLYWRITE function, 122

Precedence, 36

- summary of, 141

PRINT statement, 19

- CHARACTER, 20
- CHR, 20
- OCTAL, 21
- STRING, 21

Printers, 19

- directing program output to, 135
- SEND statement, 19

PROC, 55

Procedures, 55

- actual parameters, 56
- CALL statement, 55
- definition of, 55
- description of, 53
- END statement in, 55
- EXTERNAL, 81
- formal parameters, 56
- forward reference, 72
- functions, 62
- GOTO statement, 61
- invoking, 55
- LOCATION, 51
- names of, 55
- parameters, 55
- pass by reference, 59
- pass by value, 58
- PUBLIC, 81
- recursive, 70
- RETURN statement, 61

- RETURNS attribute, 62
 - summary of, 145
 - swapping, 71
- Program disk image, 168
- Program structure, 53, 54, 68, 73
- Program style, 54, 68
- Program termination, 105
 - EXIT, 111
 - STOP, 130
- PUBLIC storage class, 80
 - link map, 135
 - restriction of, 80
 - summary of, 140
- Push down stack, 61, 68, 69, 137
- RAM statement, 137
- Random access memory (see RAM)
- RCVDCHARACTER function, 95, 96, 123
- READ function, 89, 123, 171
- Read only memory (see ROM)
- READDATA function, 124
- Real variables (see Floating)
- RECURSIVE attribute, 70, 72
- Recursive procedures, 70
 - forward reference of, 72
 - with AUTOMATIC, 70
- Reference appendices, 99
- Registers, 91
- Relational operators, 29
 - summary of, 141
- RELOC, 82
- Relocatable binaries (see Libraries)
- Remainder function (see MOD)
- Remote computer control, 19
- Reserved words (see Keywords)
- Restricted scope (see Scope)
- RETURN statement, 61
- RETURNS attribute, 62
- ROM, 137
- ROT function, 32, 33, 125
- Rotate (see ROT)
- Sample program, 10
- Scope, 64, 69
 - definition of, 64
 - in modules, 80, 81
 - of labels, 67
 - of variables, 69
- restricting, 37
- SCSI devices, 101
- SEND statement, 19
 - CHARACTER, 20
 - CHR, 20
 - OCTAL, 21
 - STRING, 21
- Sequence number table, 135
- SET CURDEV function, 126
- Shift left (see SHL)
- Shift right (see SHR)
- SHL function, 32, 33, 127
- SHR function, 32, 33, 128
- SIN function, 129
- Sine (see SIN)
- Speed, 27
 - of arithmetic operations, 27, 29
 - of Hardware Multiply/Divide, 27, 172
 - of READ and WRITE, 89
- SQR function, 129
- Square root (see SQR)
- Statement labels, 43
- Statements, summary of, 143
- STATIC storage class, 69
 - summary of, 140
- Statistics program, 10
- STOP statement, 130
- Storage classes, 69, 80
 - AUTOMATIC, 69
 - defaults, 69
 - EXTERNAL, 81
 - PUBLIC, 80
 - STATIC, 69
 - summary of, 140
- Storage device I/O, 101, 104
 - device numbers, 102
 - EXTREAD, 113
 - EXTWRITE, 115
 - floating point data, 103
 - POLYREAD, 121
 - POLYWRITE, 122
 - READDATA, 124
 - SCSI devices, 101
 - WRITEDATA, 133
- STRING, 21
- String constants, 15
- String functions, 104
 - BYTE, 48, 109
 - PBYTE, 48, 120
- Strings, 47
 - BYTE, 48, 109
 - input of, 22

- internal format, 161
- length of, 47
- output of, 21
- PBYTE, 48, 120
- reading from, 48
- writing to, 48
- Subscripts, 45
 - out of bounds, 46, 57
- Subtraction, 25
- Swap area, 164
- SWAP attribute, 71, 72
- Swap file, 164
- SWAPINIT function, 131
- Swapping procedures, 71
 - forward reference of, 72
 - saving memory space, 71
 - speed of, 71
 - swap area, 164
 - swap file, 164
 - SWAPINIT function, 131
 - with WHEN, 94
- Symbol table, 135
- SYSLITS file, 136, 163
- System literals, 163
- System state variables, 166, 167
- Systems programming, 105
 - FIND DEVICE, 116
 - RCVDCHARACTER, 123
 - SET CURDEV, 126
 - SWAPINIT, 131
- Table of contents, 3
- TAN function, 132
- Tangent (see TAN)
- Tape drive (see Storage device I/O)
- Terminal control sequences, 20
- Terminal input, 21
 - INPUT, 21
 - LINPUT, 22
 - numeric, 21
 - strings, 22
 - summary of, 146
- Terminal interrupts, 95
- Terminal output, 19
 - PRINT, 19
 - SEND, 19
 - characters, 20
 - numeric, 19
 - octal format, 21
 - strings, 21
 - summary of, 146

- Treenames, 74, 77
- Trigonometric functions (see Arithmetic)
- TRUE, 14
- Two's complement, 159
- Unary arithmetic operators, 25
- Unresolved reference, 81
- Unsigned integers, 13, 30
- Unsigned relational operators, 30
- Variables, 13
 - declarations, 17
 - identifiers, 16
 - initialization of, 69, 80
 - storage classes of, 69, 80, 81
- Vectors (see Arrays)
- WHEN BDB14INT, 96
- WHEN BDB15INT, 96
- WHEN BREAK, 97
- WHEN D03INT, 95
- WHEN D16INT, 95
- WHEN D136INT, 95
- WHEN D140INT, 96
- WHEN DISKERROR, 97
- WHEN TTIINT, 95
- WHEN TTOINT, 95
- WHEN statement, 93
 - example of, 96
 - in modules, 79
 - INVOKE, 97
 - return from, 93
 - swapping procedures, 94
 - with GOTO, 94
- Winchester disk (see Storage device I/O)
- WRITE statement, 90, 132, 171
- WRITEDATA function, 133
- XOR, 32
 - bit operator, 32
 - truth table of, 34

